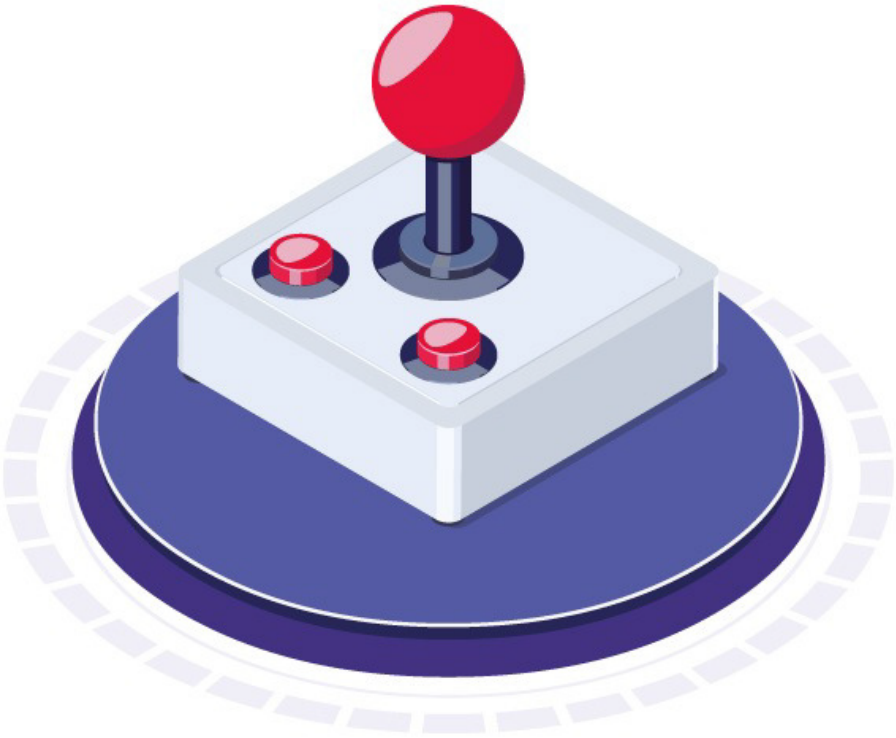
Design Patterns com C#

Aprenda padrões de projeto com os games



Casa do Código

RODRIGO GONÇALVES SANTANA

ISBN

Impresso e PDF: 978-85-7254-051-3

EPUB: 978-85-7254-052-0

MOBI: 978-85-7254-053-7

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

SOBRE O AUTOR

Rodrigo Gonçalves Santana é professor acadêmico na graduação em Sistemas de Informação, gestor de projetos, desenvolvedor e analista de softwares. Possui certificações Microsoft em desenvolvimento de software, Oracle em banco de dados e Autodesk em computação tridimensional. Formado em Sistemas de Informação, mestrando em educação, pós-graduado em Gerenciamento de Projetos com ênfase nas práticas do PMI (Project Management Institute).

Apaixonado por tecnologia e educação, tem se empenhado em aplicar, investigar, fomentar e a desenvolver tendências e metodologias educacionais aliadas a tecnologia para contribuir no processo de ensino e aprendizagem, o que levou a criar e a patentear o aplicativo "Pergunte" de reconhecimento voz e reprodução artificial da fala humana voltado à educação, como ferramenta de apoio à aprendizagem, sendo reconhecido nacional e internacionalmente. Entre outros projetos em que trabalhou, desenvolveu o aplicativo "Scout", utilizado pela seleção brasileira de Rugbi adaptado para cadeirantes nos Jogos Pan-Americanos em setembro de 2017 no Paraguai, para auxiliar na coleta de dados importantes dos atletas em quadra, onde os acertos e erros de passes, desempenho, melhores atletas em quadra, entre outras variáveis coletadas e transmitidas pelo aplicativo instantaneamente auxiliou a comissão técnica na melhor composição do time, sobre eventuais mudanças estratégicas para vencer a partida, e o Brasil ficou com medalha de bronze. E desde então, dedica-se a unir tecnologia e educação.

SOBRE O LIVRO

Talvez você esteja se perguntando o que este livro de padrões tem de diferente do padrão. Pois você já pensou em aprender conceitos e técnicas avançadas de programação com jogos como Mortal Kombat, Final Fight, Top Gear, Super Mario World, Metal Slug entre outros clássicos? Pois é, isso já torna este livro de Design Patterns diferente de qualquer outro, principalmente por ser divertido e empolgante.

Há alguns anos, leciono na graduação de Sistemas de Informação em uma disciplina destinada a explicar aspectos mais complexos de desenvolvimento de software, bem como Design Patterns. Ao longo desses anos, tenho notado que os livros de padrões de projeto de desenvolvimento de software são extremamente técnicos, o que torna difícil o entendimento de um aluno ainda na graduação, pois nem sempre ele já está no mercado de trabalho como um desenvolvedor experiente. Normalmente, a didática dos livros dessa área tende a ser bastante complicada, devido à dificuldade do assunto, e muitas vezes isso é desestimulante para o aprendizado. Logo, pensei em empreender esforços para oferecer uma forma diferente, e até lúdica, de tratar o estudo de padrões de projeto no desenvolvimento de software.

Estou muito feliz em lhe apresentar este livro, fruto de um minucioso trabalho que alinha técnica, didática e divertimento, uma harmonia de aspecto didático e metodológico. Ele foi escrito de tal forma que você possa se conectar com a proposta de cada Pattern, em um conteúdo interessante e divertido, para que a sua leitura não seja cansativa. Sobretudo, além de aprender,

propositalmente você terá um recurso que vai ajudá-lo a sempre lembrar de cada padrão de projeto: aprendizagem por meio da associação. Este livro une nossa paixão por games, que nós da área de tecnologia temos em comum. Por esse motivo, aqui você vai relembrar e conhecer jogos clássicos que vão nos ajudar a entender a proposta e como funciona cada Design Pattern, pois quando aprendemos com algo que nos remete a diversão (os games), fixamos o conteúdo facilmente.

Os capítulos foram organizados didaticamente para facilitar a evolução da sua aprendizagem. Em nosso ponto de partida, você vai adquirir uma base sólida de conceitos importantes que lhe dará o alicerce necessário para aprender qualquer padrão de desenvolvimento de software, com explicações e demonstrações para que você possa praticar e fazer junto com o livro. Uma vez que você adquiriu a base sólida, partimos para a nossa jornada de aprendizado dos Design Patterns. E, para cada Pattern estudado, você encontrará uma estrutura didática, constituída de "O Design Pattern", "Hora do desafio" e "Codificando a solução do desafio". Veja na tabela a estrutura didática que seguiremos ao estudar cada Design Pattern.

Seção	Proposta
O Design Patterns	sempre começaremos a estudar um padrão por esta seção, onde será apresentado o Design Pattern em pauta, sua classificação, bem como seu propósito.
Hora do desafio	após conhecer o Design Pattern e o seu propósito, nesta seção será apresentado um desafio de um jogo clássico que deveremos resolver com a utilização do padrão em pauta no estudo.
Codificando a solução do desafio	por fim, nesta seção é onde nós colocaremos a mão na massa e você codificará a solução para resolver o desafio com o Design Pattern que você acabou aprender, fixando a sua aprendizagem com uma atividade prática e bem explicada, comentada passo a passo.

Além do mais, o código-fonte de todos os exemplos que você vai desenvolver neste livro estão disponíveis no GitHub, organizados por Design Pattern. Você pode acessá-lo quando estiver estudando para conferir e pode até fazer download, basta acessar o seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Por fim, este livro é uma contribuição para a comunidade de desenvolvedores, com um livro em português e de fácil aprendizado sobre Design Patterns, com códigos de exemplos na linguagem C# (C-Sharp) mas que podem ser adaptados a outras linguagens. Portanto, eu espero fortemente que você goste e, sobretudo, aprenda de uma forma divertida!

Divirta-se e bons estudos!

O que eu preciso para acompanhar o livro

Uma forte característica deste livro é a prática e, com certeza, você vai desenvolver bastante código. Portanto, para que você possa estudar com este livro e colocar em prática as atividades e desafios propostos no livro, eu recomendo que você tenha instalado o Visual Studio.

Caso você ainda não tenha, recomendo a versão Community, que é um ambiente de desenvolvimento gratuito e será o suficiente para os nossos estudos neste livro. É possível fazer download da versão Community para os ambientes Windows ou MacOS em: https://visualstudio.microsoft.com/pt-br/vs/community/.
Selecione para qual ambiente (Windows ou MacOS) você deseja e

clique no botão “Baixar o VS Community 2017”. Ao terminar o download, execute o arquivo para iniciar o procedimento de instalação do Visual Studio Community.

Ao abrir, você verá a tela do Visual Studio Installer com muitas opções de pacotes de recursos que podem ser adicionados à sua instalação. Mas, para acompanhar esse livro, somente o pacote de ferramentas de Desenvolvimento para desktop com .NET será o suficiente. Selecione esta opção conforme mostrado na próxima figura, prossiga com a sua instalação e aguarde a conclusão.

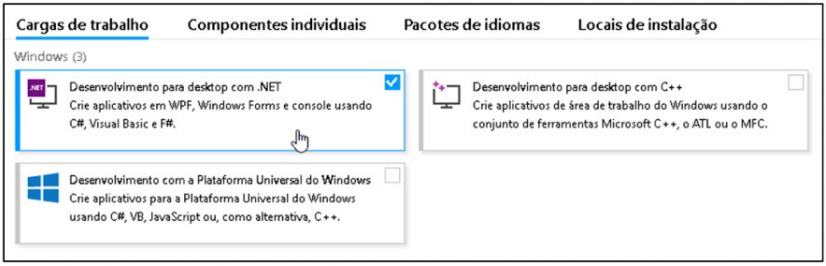

Cargas de trabalho
Componentes individuais
Pacotes de idiomas
Locais de instalação
Windows (3)
Desenvolvimento para desktop com .NET
Crie aplicativos em WPF, Windows Forms e console usando C#, Visual Basic e F#.
Desenvolvimento para desktop com C++
Crie aplicativos de área de trabalho do Windows usando o conjunto de ferramentas Microsoft C++, o ATL ou o MFC.
Desenvolvimento com a Plataforma Universal do Windows
Crie aplicativos para a Plataforma Universal do Windows usando C#, VB, JavaScript ou, como alternativa, C++.

Figura 1: Selecionando o pacote de instalação do Visual Studio Community

Pronto! Após concluir sua instalação, você está preparado(a) e agora, com o Visual Studio instalado poderá acompanhar os desafios e atividades do livro. Vamos lá!

Sumário

1 Introdução ao Design Patterns sob a perspectiva dos games 1

2 START - O ponto de partida 4

2.1 O cenário do jogo da orientação a objetos 5

2.2 Interface e classe abstrata 12

3 Os bastidores de uma instância de objeto 16

3.1 Conceitos básicos de memória 16

3.2 Construtores de instâncias 17

3.3 Método construtor de uma classe 19

4 Design Patterns - Os padrões do jogo 22

4.1 De onde surgiram? 22

4.2 Os Design Patterns clássicos 23

5 Design Patterns de Criação 26

5.1 Factory Method 26

5.2 Abstract Factory 34

5.3 Singleton 43

5.4 Builder 47

5.5 Prototype 54

6 Design Patterns de Estrutura 62

6.1 Adapter 62

6.2 Flyweight 68

6.3 Bridge 77

6.4 Composite 85

6.5 Decorator 92

6.6 Facade 98

6.7 Proxy 102

7 Design Patterns de Comportamento 111

7.1 Template Method 111

7.2 Interpreter 116

7.3 Observer 120

7.4 Visitor 127

7.5 Command 131

7.6 Strategy 137

7.7 Chain of Responsability 140

7.8 Iterator 145

7.9 Mediator 149

7.10 Memento 154

7.11 State 157

8 Conclusão 163

Capítulo 1

INTRODUÇÃO AO DESIGN PATTERNS SOB A PERSPECTIVA DOS GAMES

O filme "Pixels", lançado em julho de 2015, com um elenco de peso composto por Adam Sandler, Michelle Monaghan, Peter Dinklage, Kevin James, Josh Gad entre outros. Um filme inspirado nos videogames clássicos da década de 1980, é uma sessão nostálgica para quem viveu nessa época, e se você não é desta época, esse é um filme que eu recomendo para amantes de videogames de qualquer era, para conhecer os precursores e inspirações dos jogos atuais.

O filme mostra jogos clássicos, como o Pac-Man de 1980, que é um dos primeiros grandes símbolos do mundo dos games. O sucesso nos fliperamas de 1981, Donkey Kong contra o herói Jumpman, que tinha a missão de resgatar uma princesa enfrentando os barris arremessados por Donkey Kong. E o clássico Galaga de 1981, um dos principais games da época dos fliperamas a investir na temática de invasão espacial e o aclamado Space Invaders, de 1978, que é um dos games mais conhecidos do mundo.

Talvez aqui você esteja se perguntando o porquê da menção desse filme e o que ele tem a ver com Design Patterns, certo? Pois bem, durante o filme a frase "padrão de jogo" é repetida várias vezes pelos grandes campeões dos games, e logo no início da história, em uma sala cheia de fliperamas, o personagem Will Cooper vê Sam Brenner (seu amigo no filme) dominando o jogo Pac-Man e pergunta atônito: "Como você é tão bom nisso, se nunca jogou tanto antes?". E então, Sam responde: "Existe um padrão, eles se movem em um padrão".

Logo, conhecer os padrões é estar à frente. Pense na programação como um jogo, e você precisa conhecer as regras básicas deste jogo, como ir para frente, para trás, pular e correr etc. Se nós pensarmos no C# como um jogo, os comandos são equivalentes às estruturas da programação que nós devemos conhecer, como laços de repetições, estruturas de condições, variáveis etc. Conhecer o jogo não garante a vitória, mas conhecer os padrões nos ajuda a entender os problemas que podem aparecer durante o jogo, como evitá-los e como resolvê-los para vencer no jogo.

É para isso que existem os Design Patterns, que são padrões de desenvolvimento de software, como "truques" nos quais outros jogadores (desenvolvedores experientes) identificaram padrões no jogo (problemas recorrentes que poderemos encontrar em nossos projetos) e como solucioná-los de uma forma elegante e profissional. Ou seja, cada padrão descreve um problema em nosso ambiente e qual é a solução, de tal forma que você possa usar essa solução quantas vezes precisar [1].

Logo, nosso ponto de partida serão alguns elementos

importantes que nós precisamos conhecer e que nos darão a base necessária para aprender qualquer padrão de projeto.

Referências

[1] GAMMA, Erich et al. *Design patterns – elements of reusable object-oriented software*. Indianapolis: Addison-Wesley Professional, 1994.

Capítulo 2

START - O PONTO DE PARTIDA

Nosso ponto de partida é o paradigma da **Programação Orientada a Objetos**, também conhecida pela abreviação POO. E os grandes personagens em ação são **classes** e **objetos**.

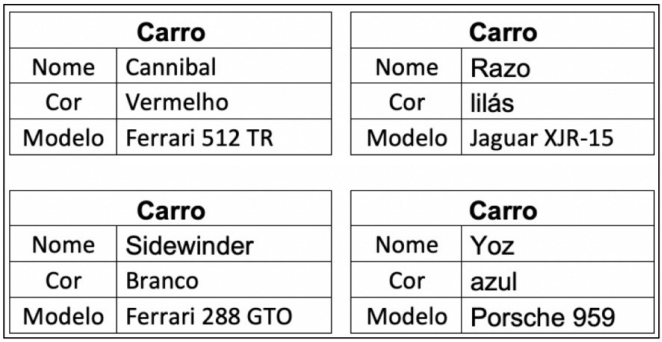
O conceito de classe

Uma **classe** é uma descrição de um conjunto de objetos com características semelhantes. Por exemplo, o Top Gear, um clássico jogo de corrida lançado em 1992 para o Super NES, contava com quatro opções de carros disponíveis para jogar: Cannibal, o carro vermelho inspirado na Ferrari 512 TR; Sidewinder, o carro branco inspirado na Ferrari 288 GTO; Razo, o carro lilás inspirado no Jaguar XJR-15; por fim, Yoz, o carro azul, inspirado no lendário Porsche 959.

Independente do carro, todos possuem nome, cor e modelo. Assim, temos uma classe que é a representação abstrata de carro. Ou seja, a classe `carro` contém uma representação das características que um carro deve ter no jogo.

O conceito de objeto

Um **objeto** representa uma entidade concreta, então os carros disponíveis no jogo (azul, branco, lilás e vermelho) são objetos da classe `carro`. Veja na ilustração a seguir quatro objetos da classe `carro`. Temos uma única classe que define as informações que um carro deve ter e quatro objetos do tipo carro.



Carro	
Nome	Cannibal
Cor	Vermelho
Modelo	Ferrari 512 TR

Carro	
Nome	Razo
Cor	lilás
Modelo	Jaguar XJR-15

Carro	
Nome	Sidewinder
Cor	Branco
Modelo	Ferrari 288 GTO

Carro	
Nome	Yoz
Cor	azul
Modelo	Porsche 959

Figura 2.1: Quatro carros derivados da classe carro

Além das classes e objetos, é importante você conhecer os pilares do paradigma de desenvolvimento orientado a objetos, que são: abstração, encapsulamento, herança e polimorfismo. Vamos compreender cada um deles a seguir.

2.1 O CENÁRIO DO JOGO DA ORIENTAÇÃO A OBJETOS

A **abstração** é o desafio de identificar características de algo do mundo real e transformá-las em uma classe. Voltando ao nosso exemplo do jogo Top Gear, quando identificamos as características de nome, cor e modelo nós praticamos abstração, ou seja,

abstraímos características de um carro. E você, consegue citar mais alguma? Se sim, você acabou de abstrair uma nova característica de um objeto do mundo real, em nosso exemplo, um carro.

A **herança** é fazer com que uma classe possa herdar características de outra. O jogo Top Gear contava com 32 pistas, em localidades famosas espalhadas pelos cinco continentes e cada pista tinha suas particularidades, mas elas também tinham características em comum. Sendo assim, o conceito de herança pode ser aplicado aqui para compartilhá-las. Veja na figura a seguir que temos duas pistas, uma de treino e outra de torneio e, embora cada pista tenha sua particularidade (a de treino não tem arquibancada, mas tem obstáculos e a de torneio não tem obstáculos, mas tem arquibancada), elas compartilham duas características em comum: nome e período. Assim, tanto a classe de pista de treino quanto a de torneio podem herdar características de "uma classe pai" chamada `pista` compartilhando o que têm em comum.

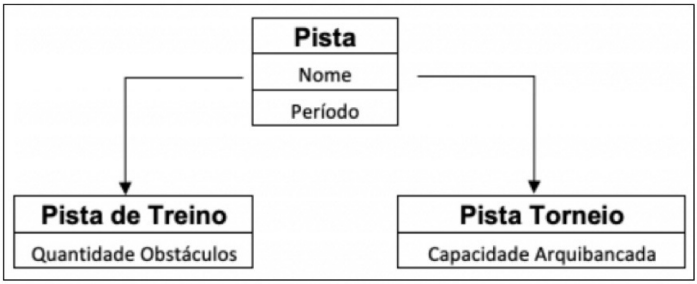


Figura 2.2: Exemplo de herança

Há também o conceito de **herança de implementação**, que permite acessar os métodos da classe herdada para reutilizá-los. Basta utilizar a palavra-chave *override* para reescrever suas

funcionalidades [2]. Veja um exemplo no código-fonte a seguir, onde temos uma classe chamada `Carro` com um método chamado `VelocidadeMaxima`, que retorna por padrão o valor `200`.

```
public class Carro
    {
        abstract public int VelocidadeMaxima()
        {
            return 200;
        }
    }
```

Na sequência, temos a classe chamada `Ferrari512TR` que herda o método `VelocidadeMaxima` da classe `carro` e com o modificador *override* sobrescrevemos o valor, então agora o valor que será retornado é `314`. Assim, utilizamos aqui o conceito de "herança de implementação". Fácil, não é?

```
public class Ferrari512TR : Carro
    {
        public override int VelocidadeMaxima()
        {
            return 314;
        }

    }
```

Outros dois conceitos importantes ligados à herança, que envolvem classes, são: **agregação** e **composição**.

Para os entendermos, é importante estabelecer o que é dependência. Portanto, pense comigo que, entre diversas peças que compõem um carro, temos o motor, volante e pneus, e sem estes elementos não é possível dirigir o carro, certo? Ou seja, dependemos deles! Desse modo, dependência é a ligação entre dois ou mais elementos. Contudo, existem dependências consideradas

fracas e fortes.

Uma dependência **forte** é chamada de **composição**, que é quando todos as partes são necessárias para compor uma classe. No exemplo do carro, há uma relação forte entre as peças motor, pneus e volante; pois elas compõem um carro e são necessárias para que seja possível dirigir o veículo. Por isso, dizemos que existe uma relação de dependência forte entre estes itens ou composição.

Já uma dependência fraca é chamada de **agregação**, que é quando a ausência das partes não influência na constituição de uma classe. Por exemplo, vamos pensar em um maratonista que existe independente de participar de alguma competição ou não.

Ainda falando sobre classe, é importante definir o nível de acesso que uma classe tem para oferecer em sua aplicação. Portanto, vale a pena compreender o conceito de **encapsulamento** que está presente no Paradigma da Orientação a Objetos, cuja ideia é basicamente ocultar o comportamento interno de uma classe para que o desenvolvedor que vai usá-la não precise se preocupar com a forma como ela funciona por dentro. Sendo assim, vamos tomar como exemplo o clássico dos jogos de luta Mortal Kombat!

O primeiro jogo da série foi lançado em 1992 para as maquinas de Arcade, você já jogou esse épico? Nesse jogo, no final da luta, o personagem abatido fica por alguns segundos sonso, sem reação, e o jogador vencedor tem esses poucos segundos para rapidamente fazer uma combinação de teclas para executar um golpe "Fatality", que é um poder especial do personagem para finalizar o adversário. Um dos meus favoritos é o personagem Liu Kang que, ao chegar próximo ao inimigo neste momento do jogo, bastava executar a sequência de teclas no controle "baixo, baixo, para trás,

frente, frente", que ele dá o especial chamado "Sore Throat", seu golpe fatal. De arrepiar!

A ideia aqui é pensar que, para executar o poder especial, basta executar a sequência de comandos e o personagem faz tudo sozinho. Nós assistimos o Fatality do personagem em execução, sem precisarmos nos preocupar com como o personagem faz toda a sequência do seu poder, porque isso está encapsulado.

Um objeto encapsulado tem dois métodos de acesso, um de leitura (`get`) e o outro de gravação (`set`). O método **get** é executado quando a propriedade de um objeto é lida. Se, porventura, uma propriedade for definida sem o acessador `get` , ela será considerada como somente gravação, já que não foi disponibilizado o conteúdo. Já o método **set** é executado quando um novo valor é atribuído à propriedade de um objeto. Sendo assim, quando uma propriedade é definida sem o acessador `set` , ela será considerada como somente leitura, pois não foi disponibilizada a propriedade que permite modificar o conteúdo do objeto (`set`) [3].

Suponha que seja necessário atualizar o poder especial do personagem Liu Kang acrescentando mais um golpe. Será necessário utilizar o método `set` , que permitirá atualizar o conteúdo do atributo referente ao poder especial. Ou, caso seja necessário visualizar a sequência de golpes do Fatality, será o método `get` que permitirá obter o conteúdo deste poder.

Outro pilar da POO é o **polimorfismo** que, de acordo com o seu significado semântico, refere-se à capacidade de assumir formas diferentes [4]. De igual modo, na programação, é a característica de um objeto ser flexível o bastante para conseguir

comportar-se de muitas formas [5]. Para exemplificar, tomaremos novamente o jogo Mortal Kombat, que conta com um lutador chamado Shang Tsung cujo poder é transformar-se em qualquer outro personagem do jogo durante a partida (característica de polimorfismo) sendo capaz de executar os poderes deles. Logo, o primeiro passo é criar uma classe base que chamaremos de `Personagem`, veja a seguir.

```
public class Personagem
{
    public virtual void Atacar() { }
}
```

No código anterior, criamos um método chamado `Atacar` sem comportamento e retorno e definimos que este método é virtual. A palavra-chave *virtual* é usada para permitir que classes derivadas substituam o comportamento do método [6]. Agora, criaremos duas classes para representar o personagem Liu Kang e outra para o Sub Zero, veja a seguir.

```
public class LiuKang : Personagem
    {
        public override void Atacar()
        {
            Console.WriteLine("Ataque do Liu Kang");
        }
    }


public class SubZero : Personagem
    {
        public override void Atacar()
        {
            Console.WriteLine("Ataque do Zub Zero");
        }
    }
```

Note que ambas herdam a classe `Personagem` e sobrescrevem

o método `Atacar`, uma vez que cada personagem tem o seu próprio poder (simplificado em um retorno escrito no console). Agora, criaremos a classe com a capacidade de assumir diferentes comportamentos, que vai representar o Shang Tsung.

```
public class ShangTsung : Personagem
    {
        public override void Atacar()
        {
            var personagens = new List<Personagem>
            {
                new LiuKang(),
                new SubZero()
            };

            foreach (var personagem in personagens)
            {
                Console.WriteLine("Agora eu sou " + personagem.Ge
tType().Name);
                personagem.Atacar();
            }
        }
    }
```

A classe `ShangTsung` também herda a classe `Personagem` como as demais. Contudo, neste exemplo, há um laço de repetição que percorre uma lista de personagens que `ShangTsung` pode assumir (LiuKang ou SubZero) e exibe na tela o Shang executando os poderes de outro personagem (assumindo outros comportamentos). Para testar esse exemplo, basta instanciar a classe `ShangTsung` e executar o método `Atacar`, conforme a seguir.

```
 class Program
    {
        static void Main(string[] args)
        {
            ShangTsung st = new ShangTsung();
            st.Atacar();
```

```
            Console.ReadKey();
        }
    }
```

2.2 INTERFACE E CLASSE ABSTRATA

Usando **interfaces**, você pode definir o comportamento que uma ou várias classes deverão seguir, potencializando o controle de muitas heranças de classes [7]. Ou seja, uma interface contém as definições das funcionalidades que uma classe deverá seguir e implementar. Logo, você pode pensar em uma interface como um contrato, e a classe que o assina deverá seguir o que foi definido nele.

No jogo Bomberman de 1983, por exemplo, existem diversos adversários no labirinto de blocos (cenário do jogo) que possuem o objetivo de implantar bombas estrategicamente para encurralar o adversário, para que a bomba exploda e os elimine. Ganha quem sobreviver (ilustração na imagem a seguir).

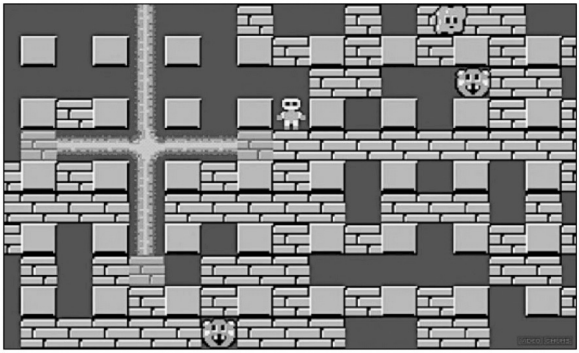


Figura 2.3: Bomberman de 1983

O jogador deve movimentar-se pelo labirinto, tendo o comportamento de andar para frente, para trás, direita ou esquerda. Assim, podemos utilizar uma interface para estabelecer os comportamentos que todos os jogadores poderão ter no jogo. Veja a seguir o exemplo da codificação dessa interface.

```
public interface Jogador
  {
      void ParaFrente();
      void ParaTras();
      void ParaDireita();
      void ParaEsquerda();
  }
```

Agora, podemos atribuir a todos os jogadores essa interface para que todos possam ter o mesmo comportamento dentro do jogo. No código de exemplo a seguir, a classe `Rodrigo` herda a interface `Jogador` criada anteriormente, e que portanto obriga o jogador `Rodrigo` a ter todos os comportamentos da interface `Jogador`.

```
public class Rodrigo : Jogador
    {
        public void ParaDireita()
        {
            Console.WriteLine("Para direita");
        }

        public void ParaEsquerda()
        {
            Console.WriteLine("Para esquerda");
        }

        public void ParaFrente()
        {
            Console.WriteLine("Para frente");
        }

        public void ParaTras()
        {
```

```
            Console.WriteLine("Para trás");
        }
    }
```

Uma **classe abstrata** refere-se a uma classe base. Esse tipo de classe não pode ser instanciado, pois ela serve de modelo para outras classes [8]. Retomando o exemplo do jogo Bomberman, o cenário é um labirinto de muitos blocos, onde cada bloco é uma classe com características comuns. Sendo assim, pode-se criar uma classe abstrata para representar as características que cada bloco deve ter. Veja o exemplo no código a seguir.

```
public abstract class Bloco
{
    public abstract int PosicaoX();
    public abstract int PosicaoY();
    public bool Explodir;
}
```

Dessa forma, todos os blocos que compõem o labirinto seguirão o modelo da classe `Bloco` para formar o cenário do jogo.

Conclusão

Neste capítulo, você aprendeu a trabalhar com classes e objetos, sobretudo, os pilares da Orientação a Objetos (abstração, herança, encapsulamento e polimorfismo). Também aprendeu a utilizar interfaces e classes abstratas. Os conhecimentos que você adquiriu aqui lhe serão uteis para sempre, como desenvolvedor ou desenvolvedora de software, em qualquer linguagem de programação orientada a objetos. Além do mais, esses conhecimentos continuarão a ser exercitados ao longo deste livro e servirão como base para você aprender qualquer Design Pattern. Veja só quanta coisa legal você já aprendeu até aqui! Convido você a continuar se aprofundando mais, nos próximos capítulos.

Referências

[2] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/override

[3] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/using-properties

[4] https://www.infopedia.pt/dicionarios/lingua-portuguesa/polimorfismo

[5] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

[6] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/virtual

[7] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/interfaces/

[8] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/abstract

CAPÍTULO 3

OS BASTIDORES DE UMA INSTÂNCIA DE OBJETO

Este é um capítulo que pode ajudar muito no desenvolvimento de aplicações para desfrutar de um bom desempenho. Aqui você vai aprender o que acontece nos bastidores de uma instância de objeto e, ao entender esses conceitos importantes, estará apto a criar aplicações olhando além do código.

3.1 CONCEITOS BÁSICOS DE MEMÓRIA

Todos os processos no mesmo computador compartilham a mesma memória física. Porém, cada um tem seu próprio espaço de endereço virtual separado. Quando estamos desenvolvendo um software, trabalhamos apenas com o espaço de endereço virtual e nunca manipulamos a memória física diretamente. Quem fica responsável por intermediar a alocação virtual e a física, no C#, é o CLR (*Common Language Runtime*), que atua como um gerenciador automático de memória [9].

Quando uma alocação de memória é solicitada, o gerenciador de memória virtual precisa localizar um único bloco livre e com espaço suficiente para atender à solicitação de alocação, uma vez que a memória virtual pode ter três estados possíveis: livre,

reservado ou comprometido. Quando se encontra no estado livre, significa que o bloco de memória não tem referências a ele e está disponível para alocação. Já quando o estado é reservado, o bloco de memória não está disponível para uso, pois já está destinado a um processo. E, por fim, se o estado está comprometido, o bloco de memória é atribuído para o armazenamento físico na memória.

3.2 CONSTRUTORES DE INSTÂNCIAS

Os construtores de instância são usados para criar e inicializar quaisquer variáveis de membro de instância. Para essa tarefa, no C#, usamos a palavra reservada *new* [10]. Tomando como exemplo o código a seguir, estamos utilizamos o comando `new` , solicitando a criação de uma nova instância chamada `meu_objeto` do objeto `bloco` na memória. Ou seja, será solicitada a alocação desse objeto em um espaço na memória suficiente para guardar todas as informações do objeto.

```
bloco meu_objeto new bloco();
```

Além do mais, o comando `new` tem a missão de criar uma referência, como uma seta, que mostra onde o objeto está armazenado na memória, para que o software que estamos criando consiga encontrá-lo e utilizá-lo. Veja a ilustração a seguir.

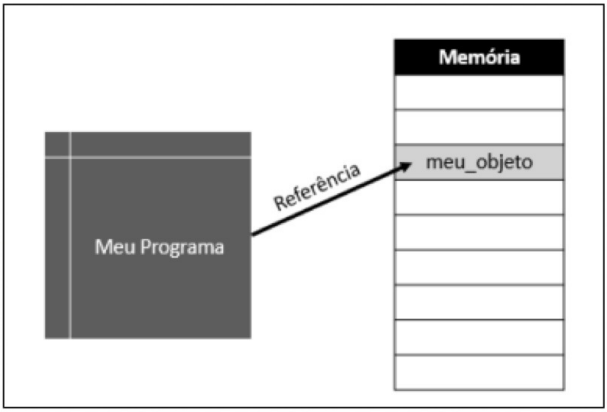


Figura 3.1: Ilustração referência de um objeto na memória

Por outro lado, quando o modificador *static* é utilizado, o objeto não precisa ser instanciado [11] e o comando `new` não é necessário. Veja no código a seguir um exemplo de utilização do modificador estático.

```
public static class bloco
    {
        public static void Faca_alguma_coisa()
        {

        }
    }
```

Desta forma, não precisamos instanciar a classe `bloco` para usar o método `faca_alguma_coisa` . Veja como seria no código a seguir a chamada ao método.

```
bloco.Faca_alguma_coisa();
```

Contudo, há uma regra importante a ser citada, pois, ao usar o modificador `static` em uma classe, todos os membros da classe deverão ser estáticos. E, embora não seja necessário instanciar um

objeto que utiliza o modificador `static`, ele também será alocado na memória tal como se usássemos a palavra-chave `new`.

3.3 MÉTODO CONSTRUTOR DE UMA CLASSE

O método construtor de uma classe sempre tem o nome da própria classe [12]. Veja por exemplo no código-fonte na sequência uma classe chamada `Construtor`.

```
public class Construtor
    {
        Construtor()
        {
            //eu sou o método construtor dessa classe.
        }
    }
```

A assinatura (nome) do método deve incluir apenas o nome do método e parâmetros (se houver) - ele não inclui um tipo de retorno. Toda vez que você instancia uma classe, esse é o primeiro método a ser executado e por isso ele se chama “construtor”, sendo disparado na construção (instanciação) da classe. Para exemplificar, vamos tomar como exemplo o épico jogo Super Mario World.

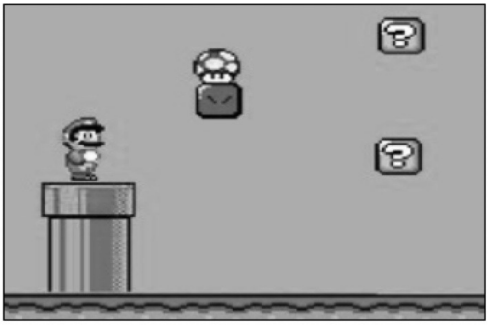


Figura 3.2: Super Mario World e um cogumelo

Quando começamos a jogar a primeira fase, o personagem Mario tem um tamanho padrão. Contudo, se ele é atingido por algo, ele fica pequeno e, se atingido novamente, perdemos o jogo. Para que o personagem Mario volte ao seu tamanho normal, precisamos capturar um cogumelo de cor vermelha no jogo, conforme ilustrado na imagem anterior. Logo, podemos definir no método construtor o tamanho inicial do Mario tendo um tamanho padrão. Veja a seguir essa codificação.

```
public class Mario
    {
        Mario()
        {
            this.Tamanho = "Normal";
        }

        public string Tamanho { get; set; }
    }
```

Posto isso, se o padrão inicial de Mario é ter o tamanho normal, poderíamos então definir o valor dessa característica no método construtor de `Mario`. Toda vez que a classe `Mario` iniciar (for instanciada) ela terá seu valor de tamanho definido como `normal`.

Conclusão

Neste capítulo, você aprendeu conceitos importantes que vão muito além de escrever código como aspectos de desempenho, lógica de funcionamento de alocação de memória e o que acontece quando um objeto é instanciado na programação. Esses conhecimentos serão cruciais para que você, como um bom desenvolvedor ou boa desenvolvedora, consiga visualizar as vantagens que os Design Patterns vão lhe oferecer quando o

assunto for desempenho. Neste ponto, você tem uma sólida bagagem e está pronto/a para mergulhar nos Design Patterns. No próximo capítulo, vamos conhecer as classificações de Design Patterns. Vejo você lá!

Referências

[9] https://docs.microsoft.com/pt-br/dotnet/standard/clr

[10] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors

[11] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/static

[12] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/constructors

Capítulo 4

DESIGN PATTERNS - OS PADRÕES DO JOGO

Neste capítulo faremos uma rápida introdução aos Designs Patterns para que você saiba qual a motivação que deu origem a estas técnicas e quais são os Design Pattern clássicos que, inclusive, você vai aprender neste livro.

4.1 DE ONDE SURGIRAM?

Esta é aquela parte do jogo em que surge uma breve história explicando o contexto do jogo. Prometo, seremos breve aqui.

MARIO
TIME
Welcome! This is Dinosaur Land. In this strange land we find that Princess Toadstool is missing again! Looks like Bowser is at it again!

Figura 4.1: Mensagem de boas-vindas no jogo Super Mario World

Certa vez, alguns programadores mais experientes perceberam que alguns problemas de desenvolvimento eram comuns em vários projetos, e que as soluções eram sempre as mesmas. Você se lembra do capítulo 1, onde foi citado que jogadores experientes descobriam o padrão dos jogos? Pois bem, quatro desenvolvedores experientes identificaram esses padrões e escreveram um livro chamado *Design Patterns: Elements of reusable object-oriented software* ou, em português, "Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos". O nome deles são Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides e, por esse motivo, ficaram mundialmente conhecido como Gang of four, o GOF, ou em português, a "gangue dos quatro".

4.2 OS DESIGN PATTERNS CLÁSSICOS

Nos relatos do GOF, encontramos os padrões de projeto clássicos. Eles são classificados em três categorias, e de acordo com o seu propósito e escopo. Veja na ilustração a seguir.

		Propósito		
		Criação	**Estrutura**	**Comportamento**
Escopo	**Classe**	Factory Method	Class Adapter	Template Method Interpreter
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweigth Proxy	Visitor Command Strategy Chair of Responsability Iterator Mediator Memento State Observer

Figura 4.2: Classificação Design Patterns clássicos

Pense na classificação como o "mapa do jogo", que nos ajuda a rapidamente encontrar ou identificar padrões olhando para a tabela.

Os padrões de **criação** se preocupam com o processo de criação de objetos. Já os **estruturais** se preocupam com a composição de classes ou de objetos. Por fim, os **comportamentais** definem a maneira como classes ou objetos deverão se comportar. Ainda, na classificação por escopo, temos os padrões com foco em classe e objetos [13].

Aqui, vamos resgatar conceitos importantes tratados no capítulo 2. Os **padrões de classe** dizem como classes devem se relacionar umas com as outras, por meio de herança. As relações, nos padrões de classe, são estabelecidas no momento da compilação.

Já os **padrões de objetos** definem os relacionamentos entre objetos e são definidos primariamente por composição. Os padrões de objetos normalmente são criados durante a execução e são muito mais dinâmicos e flexíveis.

Já a classificação **por propósito** é bem intuitiva, e agrupa os padrões com o propósito de criação, estrutura e comportamento.

Apenas um lembrete! Caso tenha alguma dúvida sobre o conceito de herança e outros trabalhados aqui, como agregação ou composição, volte ao segundo capítulo, *START - O ponto de partida*. Lá, temos a explicação destes conceitos fundamentais para você seguir o raciocínio dos Design Patterns tranquilamente.

Pois bem, agora que você passou pela parte introdutória carregando mais informações importantes em sua bagagem de desenvolvedor(a), você está pronto(a) para iniciar a jornada com Design Pattern no próximo capítulo.

Conclusão

Neste capítulo, você conheceu a origem e a classificações dos Design Patterns. Conhecer essa classificação vai ajudar a reconhecer facilmente a proposta de cada Pattern (criação, estrutura ou comportamento) e sua atuação (classe ou objeto). Portanto, no próximo capítulo você vai aprender a desenvolver os Designs Patterns de Criação.

Referências

[13] GAMMA, Erich. et al. *Design patterns – elements of reusable object-oriented software*. Indianapolis: Addison-Wesley Professional, 1994.

Capítulo 5

DESIGN PATTERNS DE CRIAÇÃO

Neste capítulo será abordado o desenvolvimento dos padrões de **criação**, que têm como foco o processo de criação de classes e/ou objetos. Será bem prático, portanto, coloque o seu Visual Studio para rodar e lá vamos nós!

5.1 FACTORY METHOD

De acordo com a tabela de classificação de Design Patterns que lhe foi apresentada no capítulo anterior, o padrão Factory Method (Método de Fábrica, em português) é classificado pelo propósito de criação e seu escopo é classe.

Também conhecido como Construtor Virtual, seu objetivo é criar uma "fábrica" de classes em tempo de execução e deixar que a classe decida seu tipo dinamicamente. Para que isso aconteça, o Factory Method dispõe uma interface, e são as subclasses que decidirão qual classe concreta deverá ser instanciada.

Guarde bem esta frase

"Abstração não deve depender de detalhes, detalhes são quem deve depender de abstrações"

Depois dessa introdução sobre o Factory Method, vamos colocá-lo em prática em um desafio!

Hora do desafio

O cenário do nosso desafio será o clássico dos jogos de luta, o Mortal Kombat de 1992. Esse jogo possui vários personagens disponíveis para serem escolhidos. Veja a ilustração da tela de seleção de personagens a seguir.

Figura 5.1: Tela de escolha de personagem no Mortal Kombat

Vale fazer uma pausa no livro, para reviver e jogar esse clássico, mas espero por você aqui, hein!

Neste clássico, existe uma grande quantidade de personagens disponíveis no jogo, contudo apenas um personagem pode ser escolhido para jogar, certo? O nosso desafio é delegar a responsabilidade para que as subclasses especifiquem o seu tipo. Ou seja, em nosso exemplo, nossa aplicação deverá ser capaz de identificar qual personagem foi escolhido para jogar e então instanciar o objeto equivalente ao personagem escolhido dinamicamente. Topa o desafio? Então vamos ao código!

Codificando a solução com Factory Method

Vamos criar como exemplo um projeto Console em C# com o nome de Factory Method.

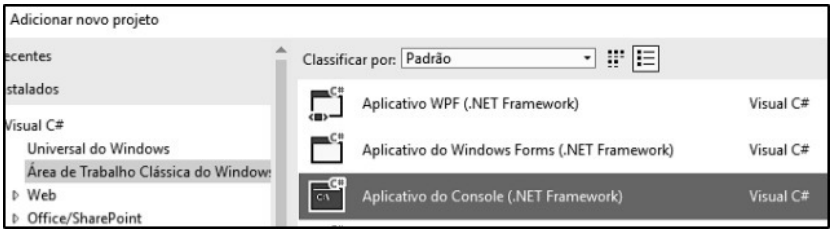


Figura 5.2: Projeto Console no C# (CSharp)

Observação importante

Seguiremos esse padrão de criação para todos os exemplos de Design Pattern deste livro. Para cada Pattern estudado será criado um projeto Console em C# e o nome do projeto será o nome do Design Pattern que estivermos estudando. Desta forma, já fica aqui combinado que esse processo inicial de criação será o mesmo para os demais projetos, não havendo a necessidade de repetir esse procedimento nas demais explicações do livro. Combinado?

Após criado o projeto, vamos agora criar uma interface chamada `IPersonagem` (botão direito em cima do nome do projeto, adicionar, novo item, e selecione `Interface`). É uma boa prática colocar a letra "I" maiúscula antes do nome para a interface a fim de ajudar a identificar os elementos de uma aplicação. Esta `Interface` vai conter a representação de um método chamado `Escolhido` e deverá ter o seguinte código:

```
public interface IPersonagem
    {
        void Escolhido();
    }
```

Agora, vamos criar os personagens concretos do jogo herdando e implementando a interface. Veja no código-fonte a seguir a criação do jogador concreto `Liu Kang`, uma classe que herda da interface `IPersonagem` e implementa o método `Escolhido` que escreve o nome do personagem na tela.

```
public class LiuKang : IPersonagem
```

```
    {
        public void Escolhido()
        {
            Console.Write("Liu Kang");
        }
}
```

Repita essa tarefa, criando mais duas classes conforme o código anterior, mudando apenas o nome da classe para os dois novos personagens: Scorpion e SubZero. Lembre-se de mudar também o conteúdo que será escrito no método `Escolhido` de cada personagem.

Agora, podemos criar a classe `Factory Method` que ficará responsável por identificar qual tipo de classe deve ser instanciada baseada no personagem que foi escolhido. Nossa classe tem um método chamado `Escolher_Personagem` com uma estrutura *switch case*, que recebe como parâmetro uma *string* `personagem` , e de acordo com o personagem identificado retorna a instância da classe correta. Veja o código completo dessa classe a seguir.

```
public class LiuKang : IPersonagem
public class FactoryMethod
    {
        public IPersonagem Escolher_Personagem(string personagem)
        {
            switch (personagem)
            {
                case "Liu Kang": return new LiuKang();
                case "SubZero": return new SubZero();
                case "Scorpion": return new Scorpion();
                default: return null;
            }
        }
    }
```

Por fim, no método `Main` da classe `Program` , podemos finalizar a codificação da nossa solução usando o padrão Factory

Method. Veja como fica o código-fonte do método `Main` :

```
static void Main(string[] args)
        {
            FactoryMethod fm = new FactoryMethod();


            Console.WriteLine("Liu Kang | SubZero | Scorpion");
            Console.WriteLine();

            Console.Write("Escolha seu Personagem: ");
            string escolha = Console.ReadLine();

            IPersonagem personagem = fm.Escolher_Personagem(escol
ha);
            Console.WriteLine();
            Console.Write("Você vai jogar com ");
            personagem.Escolhido();


            Console.ReadKey();
 }
```

Vamos agora compreender o que acabamos de codificar no método `Main` da classe `Program` . Primeiro, criamos uma instância do nosso método fábrica chamado `fm` . Em seguida, colocamos na tela as opções dos personagens para jogar (usamos três, mas você pode criar quantos personagens você achar interessante). E depois, perguntamos qual personagem será escolhido.

Até aqui, nossa aplicação não sabe qual personagem será escolhido, consequentemente a classe que deverá ser instanciada também é desconhecida. Mas, após o usuário informar com qual personagem ele deseja jogar, chamamos o método `Escolher_Personagem` , que, conforme explicado anteriormente, em uma estrutura *switch case* identificará qual foi a opção escolhida e então instanciará a classe do personagem equivalente.

Veja o resultado na imagem a seguir.

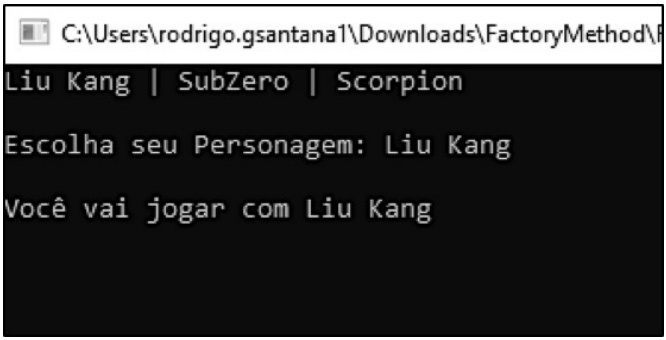

C:\Users\rodrigo.gsantana1\Downloads\FactoryMethod\
Liu Kang | SubZero | Scorpion
Escolha seu Personagem: Liu Kang
Você vai jogar com Liu Kang

Figura 5.3: Resultado do Factory Method

Para que você possa verificar que, de fato, a classe instanciada foi a classe do personagem escolhido corretamente, coloque um `Break Point` (ponto de interrupção) uma linha depois do retorno do personagem escolhido, conforme ilustrado a seguir.

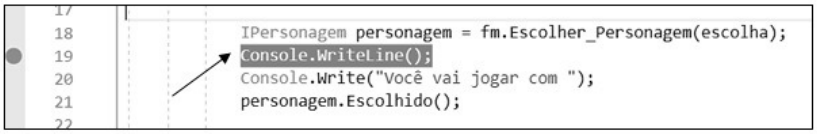


Figura 5.4: Adicionando um Break Point no código

Em seguida, execute o programa e digite o nome de um dos personagens. O programa vai parar exatamente na linha em que foi definido o `Break Point`. Agora, clique com o botão direito em `personagem` e escolha a opção **Adicionar Inspeção** conforme mostra a figura a seguir.

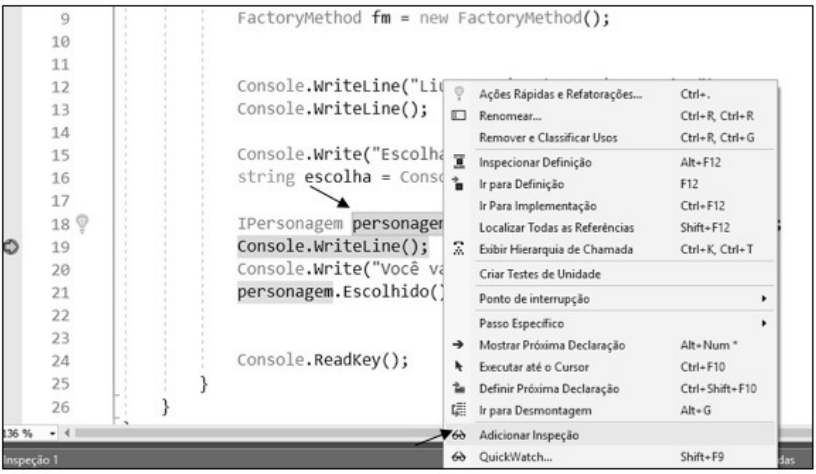


Figura 5.5: Adicionando inspeção no código

Por fim, na parte inferir do Visual Studio, é possível ver que, de fato, a classe instanciada foi `LiuKang` conforme o personagem escolhido (vide ilustração). Para praticar, tente repetir esse procedimento escolhendo outros personagens inspecionando o personagem para verificar se a classe correta do personagem foi instanciada.

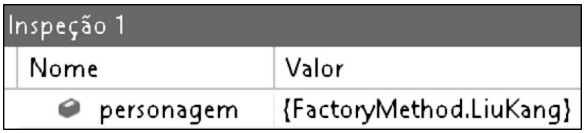


Figura 5.6: Conferindo o objeto instanciado

Agora, vamos retomar à frase inicial de que "Abstração não deve depender de detalhes, detalhes são quem deve depender de abstrações", lembra? Pois é, foi exatamente o que fizemos. Pois conseguimos resolver nosso desafio delegando a responsabilidade

para que as subclasses (personagens) especifiquem o seu tipo. Legal, não é mesmo? Estamos apenas começando, vamos para o próximo Pattern!

> **Lembrete**
>
> Você pode conferir o código-fonte completo do Design Pattern Factory Method que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:
>
> https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp
>
> Em seguida, acesse o diretório **Design Patterns de Criação** e você verá o diretório **Factory Method**.

5.2 ABSTRACT FACTORY

O **Abstract Factory** é um padrão que também pertence aos padrões com proposito de criação, e o seu escopo é objetos. O objetivo deste Design Pattern é agrupar diversas *factories* (fábricas) que possuam características semelhantes, utilizando uma única interface. Logo, convido você a um novo desafio para visualizarmos como podemos utilizá-lo.

Hora do desafio

Neste desafio, o cenário será o StarCraft, o clássico jogo de estratégia em tempo real, onde existem três raças de personagens:

os Protoss, Zergs e os Terranos. Cada raça tem suas unidades e construções diferentes, porém muito bem equilibradas.

Figura 5.7: StarCraft

Cada espécie tem sua base, que no jogo tem a função de gerar e expandir a quantidade de guerreiros. Logo, uma base é uma construção que está disponível para os três tipos de personagens, contudo, com algumas diferenças de acordo com a raça (Protoss, Zergs e os Terranos). Sendo assim, precisamos de uma estratégia para criar as bases com suas características individuais, mas também compartilhar as características que todas as bases têm em comum.

Outro aspecto deste game é que a tecnologia e as construções que estão disponíveis se expandem, evoluindo com o jogo. Então precisamos desenvolver uma solução que funcione sem que seja necessário alterar o código existente ao adicionar novos produtos ou famílias de produtos, pois a atualização acontece com muita frequência e não queremos alterar o código-fonte do jogo toda vez, certo? Diante desses desafios, vamos codificar essa solução com o

Design Pattern Abstract Factory.

Codificando a solução com Abstract Factory

Iniciaremos com a criação dos elementos que vão compor as bases no jogo: tipo de energia que sustentará a vida da base e o revestimento. Começaremos pela energia, com uma interface para representá-la. Veja o código a seguir:

```
public interface IEnergia
    {
         void Composicao();
    }
```

A interface que acabamos de criar contém apenas o método composição que será implementado em breve. Agora faremos o mesmo, criando outra interface para representar o elemento de revestimento da base. Veja a seguir o código:

```
public interface IRevestimento
    {
        void Composicao();
    }
```

Energia e revestimento são elementos que compõem uma base em nosso jogo, mas, como temos três raças diferentes de personagens, precisamos implementar essas particularidades para cada uma. Portanto, vamos definir uma classe para representar a energia da base para a raça Terran, conforme o código:

```
    public class EnergiaBaseTerran : IEnergia
    {
        public void Composicao()
        {
            Console.WriteLine("Energia de sustentação da base mec
ânica");
        }
    }
```

Perceba que a classe `EnergiaBaseTerran` herda a interface `Energia` implementando o método `Composicao` , que define a fonte de energia das bases da raça de personagens Terran. Faremos o mesmo para as outras duas raças de personagens no jogo, criando as classes `EnergiaBaseProtoss` e `EnergiaBaseZerg` como o código-fonte a seguir:

```
public class EnergiaBaseProtoss : IEnergia
    {
        public void Composicao()
        {
            Console.WriteLine("Energia de sustentação da base com
 cristais");
        }
    }


    public class EnergiaBaseZerg : IEnergia
    {
        public void Composicao()
        {
            Console.WriteLine("Energia de sustentação da base pel
a terra");
        }
    }
```

Seguindo a mesma linha de raciocínio, agora criaremos a implementação da interface revestimento, definindo a particularidade de revestimento das bases de cada raça de personagem. Acompanhe o código-fonte na sequência para a criação de revestimento para a espécie Terran.

```
public class RevestimentoBaseTerran : IRevestimento
    {
        public void Composicao()
        {
            Console.WriteLine("Base revestida pela cor verde");
        }
    }
```

Assim como já fizemos com o elemento de energia, no código anterior, nossa classe herda a interface `Revestimento` implementando o método `Composicao`, que define a característica visual das bases da raça de personagens Terran. Faremos o mesmo para as outras duas raças de personagens no jogo, criando as classes `RevestimentoBaseProtoss` e `RevestimentoBaseZerg`:

```
public class RevestimentoBaseProtoss : IRevestimento
    {
        public void Composicao()
        {
            Console.WriteLine("Base revestida pela cor amarela");
        }
    }


    public class RevestimentoBaseZerg : IRevestimento
    {
        public void Composicao()
        {
            Console.WriteLine("Base revestida pela cor vermelha")
;
        }
    }
```

Agora que temos os elementos que compõem uma base no jogo (energia e revestimento visual) e já definimos o tipo de energia e revestimento para cada base de acordo com a raça de personagens, criaremos uma interface para representar o comportamento da fábrica de bases no jogo. Vamos chamá-la de `FabricaBases` (confira o código a seguir).

```
public interface IFabricaBases
    {
        void CriarBase();
    }
```

Na sequência, vamos implementar essa fábrica que será

responsável para criar a base de acordo com o personagem solicitado dinamicamente. Veja a fábrica de bases Terran no código a seguir.

```
public class FabricaBaseTerran : IFabricaBases
    {
        public FabricaBaseTerran()
        {
            CriarBase();
        }
        public void CriarBase()
        {
            Console.WriteLine("Base Terran criada com sucesso!");

            RevestimentoBaseTerran revestimento = new Revestiment
oBaseTerran();
            revestimento.Composicao();

            EnergiaBaseTerran energia = new EnergiaBaseTerran();
            energia.Composicao();
        }
    }
```

Olhando para a fábrica de bases do personagem Terran, conforme acabamos de codificar anteriormente, você pode ver que herdamos a interface `IFabricaBases`, que nos obriga a implementar o método `CriarBase`. Neste método, definimos a instância das classes `RevestimentoBaseTerran` e `EnergiaBaseTerran` que correspondem à fábrica que montamos neste exemplo, invocando nas duas classes o método `Composicao`.

Agora, faremos o mesmo para as outras duas raças do jogo, criando as classes `FabricaBaseProtoss` e `EnergiaBaseZerg`. Veja a seguir:

```
public class FabricaBaseProtoss : IFabricaBases
    {
        public FabricaBaseProtoss()
```

```
        {
            CriarBase();
        }

        public void CriarBase()
        {
            Console.WriteLine("Base Protoss criada com sucesso!")
;

            RevestimentoBaseProtoss revestimento = new Revestimen
toBaseProtoss();
            revestimento.Composicao();

            EnergiaBaseProtoss energia = new EnergiaBaseProtoss()
;
            energia.Composicao();
        }
    }

public class FabricaBaseZerg : IFabricaBases
    {
        public FabricaBaseZerg()
        {
            CriarBase();
        }

        public void CriarBase()
        {
            Console.WriteLine("Base Zerg criada com sucesso!");

            RevestimentoBaseZerg revestimento = new RevestimentoB
aseZerg();
            revestimento.Composicao();

            EnergiaBaseZerg energia = new EnergiaBaseZerg();
            energia.Composicao();
        }
    }
```

Por fim, podemos finalizar o desenvolvimento da solução com o Abstract Factory, codificando o método `Main` da classe `Program` com uma estrutura *switch case* que ficará responsável por identificar qual raça de personagem foi escolhida para jogar e,

então, de acordo com a opção do jogador, fabricar a base correta. Vamos lá, confira a codificação:

```
    class Program
    {
        static void Main(string[] args)
        {
            IFabricaBases fabrica;
            Console.Write("Escolha um dos personagens: 1-Protoss
| 2-Zergs | 3-Terranos: ");

            switch (Console.ReadLine())
            {
                case "1":
                    fabrica = new FabricaBaseProtoss();
                    break;
                case "2":
                    fabrica = new FabricaBaseZerg();
                    break;
                case "3":
                    fabrica = new FabricaBaseTerran();
                    break;
            }

            Console.ReadLine();
        }
    }
```

Agora, podemos executar o programa e escolher, por exemplo, a raça de personagem Terran. Você verá que é montada a base equivalente ao personagem selecionado (base revestida pela cor verde e energia de sustentação com base mecânica) conforme ilustrado na imagem a seguir.

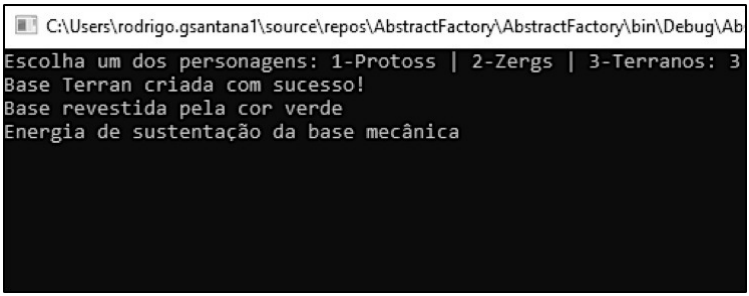

C:\Users\rodrigo.gsantana1\source\repos\AbstractFactory\AbstractFactory\bin\Debug\Ab
Escolha um dos personagens: 1-Protoss | 2-Zergs | 3-Terranos: 3
Base Terran criada com sucesso!
Base revestida pela cor verde
Energia de sustentação da base mecânica

Figura 5.8: Escolhendo a raça de personagem no jogo

Para brincar e continuar explorando o Abstract Factory, você pode criar outras raças de personagens, imaginar quais características de base teriam e, então, criar fabricas para essas bases. Que tal?

Sendo assim, mais um desafio foi concluído para o seu legado nessa jornada. Você desenvolveu elementos (ingredientes) separadamente que compõem uma base, e depois definiu as características de cada base de acordo com a raça de personagem no jogo. Por fim, criou um algoritmo flexível e capaz de montar ("fabricar") a base de acordo com as características de cada personagem. E, se no futuro, novos elementos surgirem com a expansão e atualização do jogo, basta adicioná-los como um novo elemento, bem como criamos para energia e revestimento.

Vejo você no próximo Design Pattern. Até lá!

Lembrete

Você pode conferir o código-fonte completo do Design Pattern Abstract Factory que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Criação** e você verá o diretório **Abstract Factory**.

5.3 SINGLETON

De acordo com a tabela de classificação de Design Patterns, este padrão tem o propósito de criação e seu escopo é objeto. E se você pesquisar pela sua tradução, encontrará algo como: "único, isolado, solteiro ou avulso", o que nos dá uma boa pista da sua utilidade. Ou seja, o objetivo desse padrão é garantir que apenas uma única instância de um objeto seja criada.

Hora do desafio

Neste ponto da nossa jornada, nos deparamos com alguns desafios. O primeiro deles é que precisamos controlar o acesso às instâncias da classe, garantindo que um objeto seja instanciado apenas uma vez. Segundo, reduzir a utilização de memória e, por fim, o terceiro desafio é fornecer mais flexibilidade do que com

estruturas estáticas para instanciar objetos. Complicado?

Para exemplificar, vamos pensar em um jogo de futebol. Como bem sabemos, dentre as regras está que apenas uma bola é permitida em campo. Imagine a confusão se surgisse mais de uma bola durante o jogo. Portanto, nosso desafio aqui será garantir que apenas uma bola esteja em jogo no campo.

Podemos usar o Design Pattern Singleton para resolver o problema e superar esse desafio. Pense na bola como um objeto e precisamos garantir que haja apenas uma instancia dele no campo. Vamos ao código.

Codificando a solução com Singleton

O primeiro passo é criar uma classe chamada `Singleton` em nosso projeto, e com ela vamos criar nosso objeto `bola`. Nessa classe, vamos definir uma operação responsável por garantir e manter sua própria instância única. Veja o código a seguir:

```
public sealed class Singleton
    {
        private static Singleton instancia = null;

        public static Singleton GetInstancia
        {
            get
            {
                if (instancia == null)
                {
                    instancia = new Singleton();
                    Console.WriteLine("Bola em Jogo");
                }

                return instancia;
            }
        }
```

```
        public void Mensagem(string msg)
        {
            Console.WriteLine(msg);
        }
    }
```

Aqui, vale destacar que o modificador de acesso `sealed` (selada) impede que outras classes herdem a nossa classe [14]. Você vai se deparar com um atributo `static` (estático) chamado `instancia`, que se inicia com o valor nulo. Um atributo estático pode ser acessado sem a necessidade de instanciá-lo [15].

O método `GetInstancia` dessa classe, por meio do método `get`, verifica se o atributo `instancia` tem o valor nulo. Se sim, instancia nosso objeto pela primeira vez (que é a "nossa bola"), armazena-o em memória e exibe a mensagem "Bola em Jogo" na tela. Caso a instância não seja nula, significa que o objeto já foi instanciado (ou, que "a bola já está em jogo"), sendo assim, não será instanciado novamente, retornando o objeto que já está lá. Por fim, há um método `void` (métodos `void` não retornam nada) chamado `Mensagem` para escrever na tela.

Agora que nossa classe `Singleton` está pronta, podemos retornar ao método `Main` (principal) e codificar conforme a seguir:

```
using System;

namespace Singleton
{
    class Program
    {
        static void Main(string[] args)
        {
            Singleton jogador_1 = Singleton.GetInstancia;
            jogador_1.Mensagem("Jogador 1: A bola está comigo no
meio do campo.");
```

```
            Singleton jogador_2 = Singleton.GetInstancia;
            jogador_2.Mensagem("Jogador 2: recebeu a bola.");

            Singleton jogador_3 = Singleton.GetInstancia;
            jogador_3.Mensagem("Jogador 3: recebeu o lançamento n
a linha de fundo.");

            Console.ReadKey();
        }
    }
}
```

Veja que em nenhum momento estamos instanciando o jogador no método principal do nosso projeto. Criamos três jogadores e todos eles solicitam a bola (Singleton) executando o método `Mensagem` . Pense na classe Singleton como a bola do jogo, apenas uma bola está em campo, então apenas uma instância do objeto está na memória, mesmo sendo solicitada três vezes. Ao executar o código, é possível visualizar o resultado conforme a imagem a seguir.

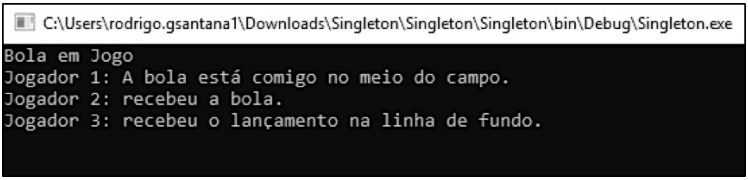

C:\Users\rodrigo.gsantana1\Downloads\Singleton\Singleton\Singleton\bin\Debug\Singleton.exe
Bola em Jogo
Jogador 1: A bola está comigo no meio do campo.
Jogador 2: recebeu a bola.
Jogador 3: recebeu o lançamento na linha de fundo.

Figura 5.9: Resultado do projeto com Singleton

Desta forma, você conseguiu resolver o desafio de garantir apenas uma bola no campo com o Singleton. Parabéns!

LEMBRETE

Você pode conferir o código-fonte completo do Design Pattern Singleton que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Criação** e você verá o diretório **Singleton**.

5.4 BUILDER

Continuando a nossa saga explorando os padrões de criação, estudaremos agora o Design Pattern **Builder**. Esse padrão de desenvolvimento pode ser utilizado na construção de objetos complexos, com uma abordagem de desenvolvimento "por partes". De acordo com a definição do GoF, o objetivo desse padrão é "separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações" [16].

Portanto, esse será o seu desafio a seguir. Criar um objeto complexo por partes!

Hora do desafio

O glorioso jogo Medal of Honor é um clássico game na

modalidade em primeira pessoa, também conhecido pela sigla FPS (*First Person Shooter*, ou em português, tiro em primeira pessoa), que tem como cenário a Segunda Guerra Mundial.

Como qualquer outro jogo de guerra, há diversos tipos de soldados que compõem um exército de batalha, como soldados de forças especiais que atacam pelo solo e pilotos de aviões de batalha que realizam ataques aéreos. Sendo assim, o nosso desafio é criar um construtor (Builder) para montar um exército inteiro para a batalha. Tudo pronto?

Codificando a solução com Builder

Primeiro, vamos criar uma classe abstrata para representar todos os tipos de soldados, sendo que todos deverão ter uma arma, um meio de transporte e um foco de ataque. O código dessa classe será o seguinte:

```
public abstract class Soldado
    {
        public string Arma       { get; protected set; }
        public string Transporte { get; protected set; }
        public string Foco       { get; protected set; }

        public abstract void EscolherArma(string arma);
        public abstract void EscolherTransporte(string transporte
;
        public abstract void DefinirFoco(string foco);
    }
```

Veja que no código da classe abstrata `Soldado` definimos três atributos com o parâmetro de acesso `protected set` (protegido), além de três métodos que possibilitarão a escolha da arma, transporte e o foco de ataque, de acordo com o tipo de soldado do exército. Desta forma, vamos agora implementar dois

tipos de soldados, criando as classes concretas `SoldadeDeInfantariaLeve` e `SoldadoDeForcasEspeciais` que herdam a classe abstrata `Soldado`.

```
public class SoldadeDeInfantariaLeve : Soldado
    {
        public override void EscolherArma(string arma)
        {
            Arma = arma;
        }

        public override void EscolherTransporte(string transporte

        {
            Transporte = transporte;
        }

        public override void DefinirFoco(string foco)
        {
            Foco = foco;
        }
   }
```

Conforme o código anterior, os três métodos recebem um valor do tipo string como parâmetro e armazenam essa informação nos respectivos atributos de `Soldado`. Veja agora, no código a seguir, a implementação da classe `SoldadoDeForcasEspeciais` que segue a mesma lógica.

```
public class SoldadoDeForcasEspeciais : Soldado
    {
        public override void EscolherArma(string arma)
        {
            Arma = arma;
        }

        public override void EscolherTransporte(string transporte

        {
            Transporte = transporte;
        }
```

```
        public override void DefinirFoco(string foco)
        {
            Foco = foco;
        }
    }
```

O próximo passo é construir uma classe abstrata para representar as classes que efetivamente vão criar cada tipo de soldado, portanto, criaremos a classe abstrata `CriadorDeSoldado`:

```
public abstract class CriadorDeSoldado
    {
        protected Soldado _soldado;

        public Soldado ObterSoldado()
        {
            return _soldado;
        }

        public abstract void Arma();
        public abstract void Transporte();
        public abstract void Foco();
    }
```

Para cada tipo de soldado, implementaremos sua respectiva classe de criação. Sendo assim, para os soldados de infantaria leve, criaremos a classe `CriadorDeInfantariaLeve` com o criador abstrato `CriadorDeSoldado`, conforme a seguir.

```
public class CriadorDeInfantariaLeve : CriadorDeSoldado
    {
        public CriadorDeInfantariaLeve()
        {
            _soldado = new SoldadeDeInfantariaLeve();
        }

        public override void Arma()
        {
            _soldado.EscolherArma("Ataque aéreo");
```

```
        }

        public override void Transporte()
        {
            _soldado.EscolherTransporte("Helicóptero de ataque do
 Exército");
        }

        public override void Foco()
        {
            _soldado.DefinirFoco("resposta rápida aérea");
        }
    }
```

No método construtor da classe `CriadorDeInfantariaLeve` instanciamos esse tipo de soldado, armazenando a instância no atributo `_soldado`. E para os métodos `Arma`, `Transporte` e `Foco` atribuímos os valores pertinentes a este tipo de soldado. Seguindo o mesmo raciocínio, no código a seguir vamos criar a classe `CriadorForcasEspeciais`, atribuindo os valores para um soldado típico das forças especiais.

```
    public class CriadorForcasEspeciais : CriadorDeSoldado
    {
        public CriadorForcasEspeciais()
        {
            _soldado = new SoldadoDeForcasEspeciais();
        }

        public override void Arma()
        {
            _soldado.EscolherArma("Fuzil");
        }

        public override void Transporte()
        {
            _soldado.EscolherTransporte("Carro de operações espec
iais");
        }

        public override void Foco()
        {
```

```
            _soldado.DefinirFoco("combate em solo");
        }
    }
```

Finalmente, podemos criar a nossa classe que ficará responsável por criar o exército de batalha no jogo. Chamaremos essa classe de `Exercito`, conforme a seguir.

```
public class Exercito
    {
        public void ConstruirSoldado(CriadorDeSoldado criadorDeSo
ldado)
        {
            criadorDeSoldado.Arma();
            criadorDeSoldado.Transporte();
            criadorDeSoldado.Foco();
        }
    }
```

Na classe `Exercito`, há o método `ConstruirSoldado`, que recebe como parâmetro a classe abstrata `CriadorDeSoldado` contendo a apresentação dos atributos que qualquer soldado precisar ter (arma, transporte e foco de ação). Agora, para concluir o desafio com a implementação do Design Pattern Builder, chegou o momento de codificar o método `Main` da classe `Program`. Vamos lá!

```
class Program
    {
        static void Main(string[] args)
        {
            var exercito = new Exercito();
            CriadorDeSoldado criadorDeSoldado;
            Soldado soldado;

            // criando um soldado das Forças Especiais
            criadorDeSoldado = new CriadorForcasEspeciais();
            exercito.ConstruirSoldado(criadorDeSoldado);
            soldado = criadorDeSoldado.ObterSoldado();
            Console.WriteLine("Soldado com as características: {0
```

```
}, {1}, {2}",
                    soldado.Arma, soldado.Transporte, soldado.Foc
o);

            // criando o soldado de Infantaria Leve
            criadorDeSoldado = new CriadorDeInfantariaLeve();
            exercito.ConstruirSoldado(criadorDeSoldado);
            soldado = criadorDeSoldado.ObterSoldado();
            Console.WriteLine("Soldado com as características: {0
}, {1}, {2}",
                    soldado.Arma, soldado.Transporte, soldado.Foc
o);

            Console.ReadKey();
        }
    }
```

No método principal da classe `Program` do projeto Builder, iniciamos instanciando exército, o criador de soldados e soldado. Em seguida, o criador de soldado instancia um soldado das forças especiais e então colocamos na tela, concatenado, qual é a arma, transporte e o foco desse tipo de soldado. A mesma coisa é feita nas próximas linha do código, mas para criar um soldado de infantaria leve.

Sendo assim, você conseguiu comprimir mais um desafio. Pois você desenvolveu um construtor (Builder) de um objeto complexo (um exército inteiro) que é formado por partes (vários tipos de soldados). Missão completa, parabéns!

LEMBRETE

Você pode conferir o código-fonte completo do Design Pattern Builder que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Criação** e você verá o diretório **Builder**.

5.5 PROTOTYPE

Para encerrar com chave de ouro os padrões de desenvolvimento com propósito de criação, exploraremos o Pattern **Prototype** (protótipo, em português). O escopo desse padrão é objetos, sendo que ele permite copiar objetos existentes sem que seu código seja dependente da classe. Quer ver como podemos utilizar esse padrão de desenvolvimento de projeto? Vamos ao desafio.

Hora do desafio

Vamos retomar a atenção para o clássico jogo Super Mario World, desta vez para as nuvens do cenário. Se você reparar conforme demonstrado na imagem a seguir indicado pelas setas, as nuvens são exatamente iguais - apenas estão em posições

diferentes no cenário do jogo.

Figura 5.10: Nuvens do jogo Super Mario World

O seu desafio será criar uma cópia exata de uma nuvem. Para tanto, utilizaremos o Design Pattern Prototype, com o qual criaremos um novo objeto da mesma classe, percorrendo todos os campos do objeto original e copiando os seus valores para o novo objeto copiado. Sendo assim, vamos codificar a solução do desafio com o Prototype.

Codificando a solução com Prototype

No primeiro passo, vamos criar uma classe abstrata para representar o molde de uma nuvem, que servirá como exemplo para as nuvens concretas.

```
public abstract class NuvemMolde
{
    public abstract NuvemMolde Clone();
```

```
    }
```

Agora, criaremos uma nuvem implementando a representação abstrata que acabamos de criar. Chamaremos essa classe de `NuvemConcreta` conforme codificado a seguir:

```
    public class NuvemConcreta : NuvemMolde
    {
        private string cor_preenchimento;
        private string cor_contorno;

        public NuvemConcreta(string preenchimento, string contorn
o)
        {
            this.cor_preenchimento = preenchimento;
            this.cor_contorno = contorno;
        }

        public override NuvemMolde Clone()
        {
            Console.WriteLine("A nuvem clonada tem contorno " + t
his.cor_contorno + " e preenchimento " + this.cor_preenchimento);
            return this.MemberwiseClone() as NuvemMolde;
        }
    }
```

Na classe `NuvemConcreta`, criamos os atributos `cor_preenchimento` e `cor_contorno`, que serão as características visuais da nuvem definidas no método construtor dessa classe. Em seguida, com a palavra-chave `override`, sobrescrevemos o método `Clone` que foi herdado da classe `NuvemMolde`, escrevendo na tela as características visuais da nuvem e realizando uma cópia dela com o `MemberwiseClone` [17].

O próximo passo é criar uma coleção para organizar os objetos para que possamos reaproveitar as clonagens. Utilizaremos um dicionário de dados com um parâmetro do tipo `string` para

identificador as nuvens clonadas. Veja a seguir, a criação e codificação da classe, que vamos chamar de `GerenciadorNuvens`.

```
public class GerenciadorNuvens
    {
        private Dictionary<string, NuvemMolde> nuvens = new Dicti
onary<string, NuvemMolde>();

        public NuvemMolde this[string key]
        {
            get { return nuvens[key]; }
            set { nuvens.Add(key, value); }
        }
    }
```

Agora, no método `Main` da classe `Program`, iniciamos com a instância da classe responsável por gerenciar a clonagem das nuvens no jogo.

```
static void Main(string[] args)
        {
            GerenciadorNuvens gerenciador_de_nuvens = new Gerenci
adorNuvens();

            gerenciador_de_nuvens["padrão"] = new NuvemConcreta("
branco", "azul");
            gerenciador_de_nuvens["personalizada"] = new NuvemCon
creta("branco", "laranja");

            NuvemConcreta um   = gerenciador_de_nuvens["padrão"].
Clone() as NuvemConcreta;
            NuvemConcreta dois = gerenciador_de_nuvens["padrão"].
Clone() as NuvemConcreta;
            NuvemConcreta tres = gerenciador_de_nuvens["personali
zada"].Clone() as NuvemConcreta;

            Console.ReadKey();
        }
```

Em seguida, criamos duas nuvens nomeando suas chaves como `padrão` e `personalizada`. Esses dois objetos, que representam

dois tipos de nuvens, serão armazenados no dicionário que criaremos, e o identificador de cada objeto será o tipo que definimos para cada nuvem, em nosso exemplo, `padrão` e `personalizada`. Por fim, criamos três nuvens no jogo (`um`, `dois` e `tres`) executando o método `Clone`. Sendo assim, seguindo esses passos, ao executar o programa você terá o resultado equivalente à imagem a seguir.

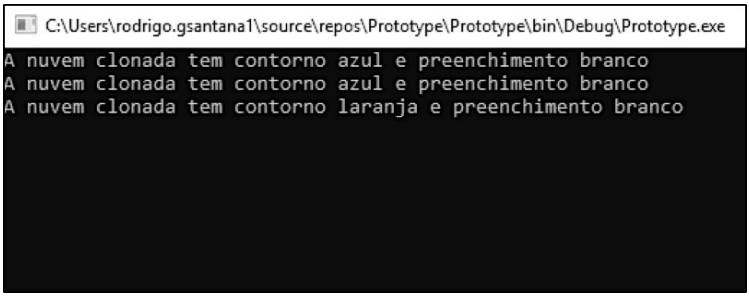

C:\Users\rodrigo.gsantana1\source\repos\Prototype\Prototype\bin\Debug\Prototype.exe
A nuvem clonada tem contorno azul e preenchimento branco
A nuvem clonada tem contorno azul e preenchimento branco
A nuvem clonada tem contorno laranja e preenchimento branco

Figura 5.11: Nuvens do jogo Super Mario World

Se você colocar um `Break Point` (ponto de parada) em uma das nuvens, iniciar novamente o projeto e inspecionar o objeto `gerenciador_de_nuvens`, você verá que dois objetos foram criados (nuvem padrão e nuvem personalizada), mas os dois tipos de nuvem são uma cópia do objeto `NuvemConcreta`.

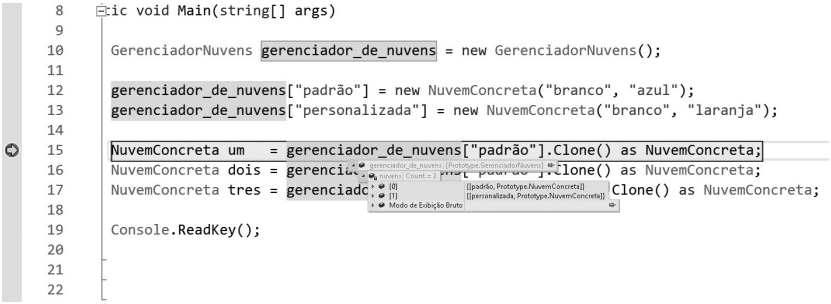

ic void Main(string[] args)
GerenciadorNuvens gerenciador_de_nuvens = new GerenciadorNuvens();
gerenciador_de_nuvens["padrão"] = new NuvemConcreta("branco", "azul");
gerenciador_de_nuvens["personalizada"] = new NuvemConcreta("branco", "laranja");
NuvemConcreta um = gerenciador_de_nuvens["padrão"].Clone() as NuvemConcreta;
NuvemConcreta dois = gerenci
Clone() as NuvemConcreta;
NuvemConcreta tres = gerenciadc
Clone() as NuvemConcreta;
Console.ReadKey();

Figura 5.12: Inspecionando o objeto gerenciador_de_nuvens

Contudo, por mais que as três nuvens sejam uma cópia (clone) do objeto `NuvemConcreta`, pudemos também personalizar a cor de contorno da terceira cópia.

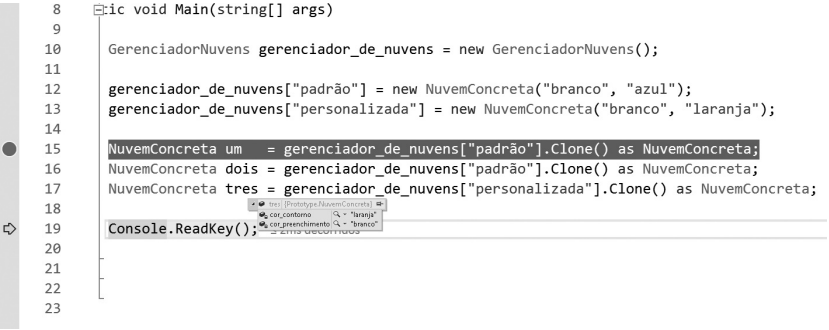

ic void Main(string[] args)
GerenciadorNuvens gerenciador_de_nuvens = new GerenciadorNuvens();
gerenciador_de_nuvens["padrão"] = new NuvemConcreta("branco", "azul");
gerenciador_de_nuvens["personalizada"] = new NuvemConcreta("branco", "laranja");
NuvemConcreta um = gerenciador_de_nuvens["padrão"].Clone() as NuvemConcreta;
NuvemConcreta dois = gerenciador_de_nuvens["padrão"].Clone() as NuvemConcreta;
NuvemConcreta tres = gerenciador_de_nuvens["personalizada"].Clone() as NuvemConcreta;
Console.ReadKey();

Figura 5.13: Inspecionando o objeto gerenciador_de_nuvens

Portanto, mais um desafio concluído. Parabéns!

LEMBRETE

Você pode conferir o código-fonte completo do Design Pattern Prototype que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Criação** e você verá o diretório **Prototype**.

Conclusão

Você conclui a saga dos Design Patterns clássicos com o propósito de criação de objetos e classes. Primeiro, foi-lhe apresentando o padrão Factory Method, que tem como escopo classes e cujo objetivo é criar uma "fábrica" de classes em tempo de execução, deixando que a classe decida seu tipo dinamicamente.

Por outro lado, o Abstract Factory tem como escopo objetos, com o objetivo de agrupar fábricas de objetos que possuam características semelhantes utilizando uma única interface. Você também conheceu o Design Pattern Singleton, que garante que apenas uma única instância de um objeto seja criada, além do Builder, o Design Pattern que é utilizado na construção de objetos complexos por partes. E por fim, você concluiu os padrões de criação com o Pattern Prototype, que clona objetos existentes sem que seu código seja dependente da classe.

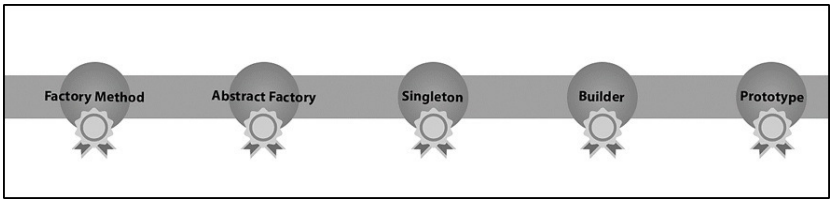


Figura 5.14: A saga dos Design Patterns de criação

Espero você, na saga dos Design Patterns de Estrutura!

Referências

[14] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/sealed

[15] https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

[16] GAMMA, Erich. et al. *Design patterns – elements of reusable object-oriented software*. Indianapolis: Addison-Wesley Professional, 1994.

[17] https://docs.microsoft.com/pt-br/dotnet/api/system.object.memberwiseclone?view=netframework-4.7.2

Capítulo 6

DESIGN PATTERNS DE ESTRUTURA

Neste capítulo, será abordado o desenvolvimento dos padrões de **estrutura**, que tem como foco estabelecer a forma como as classes e/ou objetos deverão ser estruturados. Este será mais um capítulo bem prático, portanto, com o seu Visual Studio aberto lá vamos nós!

6.1 ADAPTER

Há dois padrões Adapter dentro da categoria de Design Patterns com o propósito estrutural, sendo o adaptador de classe e de objetos. A diferença entre eles é sutil. O Adapter de classe usa herança e só pode envolver uma classe; ele não pode envolver uma interface, pois, por definição, deve derivar de alguma classe base. Por outro lado, o de objetos usa composição e pode envolver classes ou interfaces, ou ambos, pois contém, como um membro encapsulado particular, a instância de objeto de classe ou interface. Ou seja, o Adapter atua como um intermediador, tal como um adaptador é um intermediário entre tomadas de energia de padrões de pinos diferentes.

Com essa introdução do Adapter, vamos ao desafio, para

exemplificar a utilização desse padrão de projeto.

Hora do desafio

Um dos clássicos Arcade é o jogo Metal Slug X de 1999. Nesse jogo, Morden resolve se aliar a alienígenas que vêm do espaço com objetivo de conquistar o mundo. Durante as missões, seu objetivo é libertar os prisioneiros que oferecem armas e equipamentos que podem ser usados no combate para destruir os alienígenas. Além do mais, ao longo das fases, o personagem também pode obter veículos de combate, como usar um tanque de guerra ou um avião nas batalhas.

Figura 6.1: Personagem do jogo Metal Slug X entrando em um avião de batalha

O personagem pode tanto caminhar pelas fases quanto utilizar um veículo de combate, o que pode mudar completamente os comandos do jogo. Desta forma, temos aqui o desafio de desenvolver um "adaptador" que torne possível que os mesmos comandos funcionem tanto se o personagem estiver em um veículo ou não. Vamos adaptar as ações do personagem de acordo

com seu meio de locomoção no jogo.

Por exemplo, o comando “andar” tem que funcionar tanto se o personagem estiver pilotando o avião de batalha, cuja ação será “voar”, como se estiver apenas caminhando, então sua ação será “ir para frente”. Portanto, vamos codificar a nossa solução com o padrão de nome mais sugestivo para esse desafio, o Design Pattern Adapter.

Codificando a solução com Adapter

Para resolver nosso desafio, vamos começar pela codificação de uma interface que vamos chamar de `IAcao`:

```
public interface IAcao
    {
        void Andar(string jogador);
        void Atirar();

    }
```

Nesta interface, vamos definir dois métodos `void` (sem retorno). O primeiro com o nome `Andar`, que vai receber como parâmetro o nome do jogador, e outro, que será chamado de `Atirar`. Agora, podemos codificar a classe `Personagem` implementando a interface que acabamos de criar.

```
public class Personagem : IAcao
    {
        public void Atirar()
        {
            Console.WriteLine("Atirou no jogo!");
        }

        public void Andar(string personagem)
        {
            Console.WriteLine(personagem + " ANDOU PARA FRENTE!")
;
```

```
        }
    }
```

Nossa classe `Personagem` herda a interface `IAcao` e implementa os métodos nomeados `Atirar` e `Andar` exibindo uma mensagem na tela de acordo com a ação executada. Agora, vamos criar outra classe para representar o avião do nosso jogo, veja a seguir.

```
public class Aviao
    {
        public void Voar(string personagem)
        {
            Console.WriteLine(personagem + " VOOU PARA FRENTE!");
        }

        public void SoltarMissil()
        {
            Console.WriteLine("Soltou um míssil no jogo!");
        }
    }
```

A classe `Aviao` tem seus próprios métodos (`Voar` e `SoltarMissil`) que são diferentes dos métodos da classe `Personagem` (`Atirar` e `Andar`). Diante de duas situações de jogo com comportamentos diferentes, precisamos criar a classe que será responsável por implementar o Design Pattern Adapter, veja a seguir.

```
public class Adapter : IAcao
    {
        Aviao aviao;

        public Adapter(Aviao novo_aviao)
        {
            this.aviao = novo_aviao;
        }

        public void Andar(string jogador)
        {
```

```
            this.aviao.Voar("Rodrigo");
        }

        public void Atirar()
        {
            this.aviao.SoltarMissil();
        }
    }
```

Observe que o nosso adaptador implementa a interface `IAcao`, porém adaptando cada método para que o personagem execute as ações do avião. Veja que, para o método `Andar` do personagem, é executado o método `Voar` que é uma ação do avião. Da mesma forma com o método `Atirar` de personagem que é executado o método `SoltarMissil`, que é um comportamento do avião. Agora, vamos finalizar nossa codificação no método principal do nosso programa, conforme a seguir.

```
static void Main(string[] args)
        {
            Personagem rodrigo = new Personagem();
            Aviao aviao_de_batalha = new Aviao();

            IAcao adaptador = new Adapter(aviao_de_batalha);

            Console.WriteLine("--- CAMINHANDO ---");
            rodrigo.Andar("Rodrigo");
            rodrigo.Atirar();

            Console.WriteLine();

            Console.WriteLine("--- PEGOU UM AVIÃO NO JOGO ---");
            adaptador.Andar("Rodrigo");
            adaptador.Atirar();

            Console.ReadKey();
        }
```

Em nosso método `Main`, primeiramente, instanciamos os objetos `Personagem` e `Aviao`. Na sequência, criamos um

adaptador com o tipo da interface `IAcao`, que recebe a instância do adaptador para `aviao_de_batalha`. Além de escrever a mensagem "CAMINHANDO", executamos o método `Andar` passando o nome do personagem "Rodrigo" e o método `Atirar`. Depois, escrevemos na tela que o jogador pegou um avião no jogo, e então, executamos os métodos `Andar` e `Atirar` novamente, mas agora são ações do avião. Veja o resultado logo a seguir.

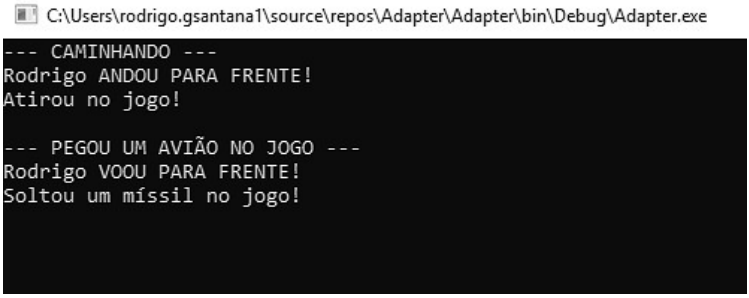


Figura 6.2: Resultado do projeto Adapter

Note que chamamos os mesmos métodos (`Andar` e `Atirar`) tanto para o objeto `rodrigo` quanto para o `adaptador`. A grande "sacada aqui", está nessa classe `adaptador`. Pois, quando chamamos na `Program` o `Andar`, internamente a classe adaptador executa o método `Voar` da classe `Aviao`. A mesma coisa acontece quando chamamos na `Program` o método `Atirar`; internamente programamos a classe `adaptador` para chamar o método `SoltarMissil` da classe `Aviao`. Ou seja, adaptamos!

Deste modo, conseguimos concluir o desafio proposto. Desenvolvemos um adaptador para adaptar os mesmos comandos de jogo tanto se o personagem estiver em um avião ou caminhando no jogo. Ponto para você, parabéns!

Lembrete

Você pode conferir o código-fonte completo do Design Pattern Adapter que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Adapter**.

6.2 FLYWEIGHT

Passaremos agora a explorar o padrão Flyweight, que é classificado com o propósito estrutural e de escopo objeto. Como ponto de partida, olhar para a tradução do nome desse padrão pode nos ajudar a entender o seu objetivo: seria algo como "peso-mosca", o que pode nos levar a pensar em algo tão leve quanto o peso de uma mosca. Será que isso realmente faz sentido?

Pois bem, o objetivo desse padrão é melhorar o desempenho compartilhando objetos com características parecidas. Ele provê recursos para que objetos existentes possam ser reaproveitados, modificando suas propriedades conforme solicitado, em vez da necessidade de sempre instanciar novos objetos.

Um dos requisitos não funcionais de grande importância de qualquer software é performance, e deixar o software rápido

implica em pouco consumo de memória. Você se lembra do capítulo 3? Lá, vimos o que acontece quando um objeto é instanciado na memória. Sendo assim, um bom caminho é reduzir a quantidade objetos criados, já que o processo de instanciação pode ser relativamente custoso. Esse será o nosso desafio.

Hora do desafio

O clássico Super Mario World do console Super Nintendo, que ainda marca gerações, é um dos meus preferidos. Nesse jogo, logo em uma das fases inicias, surgem várias tartarugas, conforme ilustrado na imagem a seguir.

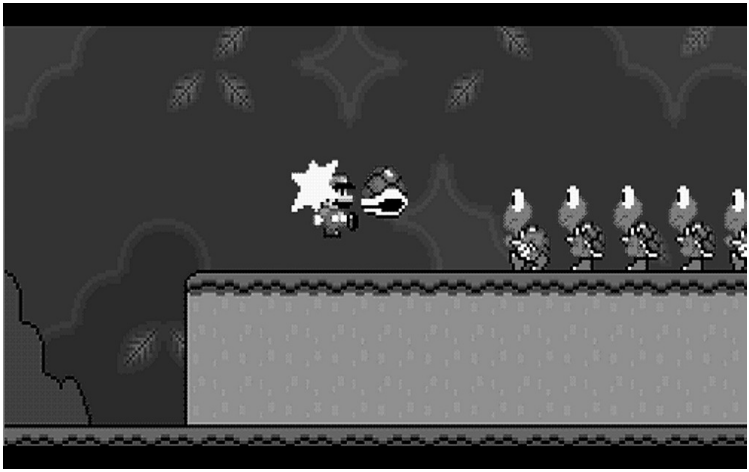


Figura 6.3: Ilustração do jogo Super Mario World

Agora, imagine que cada tartaruga é um objeto e que além de tartarugas vermelhas, é possível encontrar outras cores no decorrer do jogo. Sendo assim, vamos imaginar seis tartarugas, sendo: três vermelhas, uma verde, uma laranja e uma azul. O nosso desafio é desenvolver uma solução que garanta apenas uma instância para

cada tipo de tartaruga, para evitar sobrecarregar a memória. Imagine mil tartarugas no jogo, cada uma ocupando uma instancia na memória, que problema! Isso poderia tornar o jogo extremamente lento.

Para resolver esse desafio, podemos compartilhar objetos semelhantes que já estão instanciados e reduzir a quantidade de objetos alocados na memória. No nosso exemplo, conforme ilustrado na figura anterior, em vez de termos seis instâncias de tartarugas, poderíamos reduzir para apenas quatro, pois usando o padrão Flyweight uma única instancia da tartaruga vermelha pode ser compartilhada com os três cascos vermelhos que estão no jogo.

Sendo assim, apresento-lhe duas características de um objeto no padrão Flyweight: intrínsecas e extrínsecas. Uma característica intrínseca do objeto é uma propriedade imutável, isto é, que caracteriza o objeto compartilhado, mas que não muda (no nosso exemplo, todos os objetos são tartarugas). E uma característica extrínseca são propriedades variáveis que podem receber novos valores a cada acesso (por exemplo, cada tartaruga pode ter sua cor). Agora que temos em mente a proposta do Pattern Flyweight vamos codificá-la.

Codificando a solução com Flyweight

Primeiro, criaremos uma classe abstrata que servirá como um molde para as tartarugas, definindo as características intrínsecas (`condicao` e `acao`) e extrínsecas (`cor`). Ainda, vamos atribuir aos atributos `condicao` e `acao` o modificador de acesso `protected` estabelecendo que eles serão acessíveis somente dentro desta classe ou instâncias derivadas da classe [18]. E, por

fim, um método chamado `Mostra` que ao ser implementado exibirá a tartaruga na tela; esse método é `void` e não possui nenhum tipo de retorno [19].

```
public abstract class Tartaruga
    {
        protected string condicao;
        protected string acao;
        public    string cor { get; set; }

        public abstract void Mostra (string cor);
    }
```

Agora que temos o molde pronto do nosso objeto, podemos implementar as classes concretas de tartaruga. Veja no código-fonte a seguir a criação da classe `Vermelha` , que implementa uma `Tartaruga` concreta ao herdar a classe abstrata `Tartaruga` .

```
public class Vermelha : Tartaruga
    {
        public Vermelha()
        {
            this.condicao = " tartaruga dentro do casco, ";
            this.acao = "rodando no chão - ";
        }

        public override void Mostra(string qualcor)
        {
            this.cor = qualcor;
            Console.WriteLine(condicao + acao + cor.ToUpper());
        }
    }
```

No código, você pode observar que definimos a `condicao` da tartaruga (do tipo vermelha), que em nossa ilustração definimos como "tartaruga dentro do casco" e para `acao` atribuímos "rodando no chão".

Ação e condição

Imagine que as tartarugas do jogo podem esconder-se dentro dos seus próprios cascos. Sendo assim, essa é uma **condição** em que você pode encontrar uma tartaruga.
Agora, pense que, além de uma tartaruga estar dentro do seu casco, ela possa girar para atacar. Essa é a **ação**.

Já a implementação do método chamado `Mostra` solicita uma variável do tipo string `qualcor` , armazena a cor da tartaruga (característica extrínseca, mutável) e então escreve na tela a condição, ação e a cor da tartaruga. Aqui, vale destacar a utilização do recurso `this` . Essa palavra-chave refere-se à instância atual da classe [20]. Quando definimos `this.cor` , estamos referindo-nos à `cor` do próprio objeto.

Por ora, temos a classe abstrata que é o nosso molde de tartaruga e uma classe concreta chamada `Vermelha` , que representa uma tartaruga do tipo vermelha. Seguindo o exemplo anterior, crie mais três classes representando cores de tartarugas azul, laranja e verde. Em seguida, vamos codificar a classe que será responsável por gerenciar as instâncias e compartilhar com objetos de características semelhantes. Acompanhe o código-fonte a seguir.

```
using System;
using System.Collections.Generic;

namespace Flyweight
{
    public class Flyweight
```

```
    {
        private Dictionary<string, Tartaruga> lista_de_tartarugas
 = new Dictionary<string, Tartaruga>();

        public Tartaruga GetTartaruga(string cor)
        {
            Tartaruga t = null;

            if (lista_de_tartarugas.ContainsKey(cor))
            {
                t = lista_de_tartarugas[cor];
            }
            else
            {
                switch (cor)
                {
                    case "azul":     t = new Azul();     break;
                    case "verde":    t = new Verde();    break;
                    case "laranja":  t = new Laranja();  break;
                    case "vermelha": t = new Vermelha(); break;
                }

                lista_de_tartarugas.Add(cor, t);
            }

            return t;
        }
    }
}
```

Em nossa classe `Flyweight`, primeiramente, criamos um objeto chamado `lista_de_tartarugas` do tipo `Dictionary` que depende da importação de `System.Collections.Generic` (importado a classe por meio da palavra reservada `using`). Um objeto `Dictionary` representa uma coleção de chaves e valores [21]. Em nosso caso, a chave será a `cor` e o valor será o objeto `Tartaruga`. Na sequência, criamos um método chamado `GetTartarugas` que recebe o parâmetro `cor`.

O método `GetTartarugas`, utilizando uma estrutura

condicional, é capaz de verificar se já existe uma tartaruga instanciada com as mesmas características (em nossa ilustração, da mesma cor). Caso não exista, a nova instância da tartaruga é registrada no dicionário `lista_de_tartarugas`. Contudo, se já existe na lista, não é instanciada uma nova tartaruga, simplesmente vamos reaproveitar a que temos e está na memória com características semelhantes. Legal, não é mesmo?

Agora que temos a classe responsável pelo funcionamento da lógica do Design Pattern Flyweight pronta, vamos à classe `Program` e no método `Main` finalizar a nossa solução. Veja no código a codificação:

```
static void Main(string[] args)
        {
            Flyweight flyweight = new Flyweight();
            string cor = string.Empty;

            Tartaruga tartaruga;

            while (true)
            {
                Console.WriteLine();

                Console.Write("Qual tartaruga enviar para tela: "
);
                cor = Console.ReadLine();

                tartaruga = flyweight.GetTartaruga(cor);
                tartaruga.Mostra(cor);

                Console.WriteLine();
                Console.WriteLine("------------------------");
            }
        }
```

Conforme podemos ver na classe `Program`, primeiro instanciamos nossa classe `Flyweight`. E dentro, criamos um laço

de repetição que recebe o parâmetro `true` (verdadeiro), para que tenhamos um laço de repetição sem fim até que o programa seja interrompido (fechado) e que perguntará qual tartaruga enviar para a tela. De acordo com a tartaruga solicitada, o método `GetTartaruga` da classe `Flyweight` vai verificar se já existe um objeto instanciado com as mesmas características que possa ser compartilhando, evitando uma nova instância - caso não haja, instanciar o novo objeto conforme já vimos anteriormente, e então, exibi-lo na tela.

Agora, vamos testar a nossa solução solicitando três tartarugas vermelhas, uma verde, uma laranja e uma azul. E teremos no total seis tartarugas na tela, conforme é possível ver na imagem a seguir.

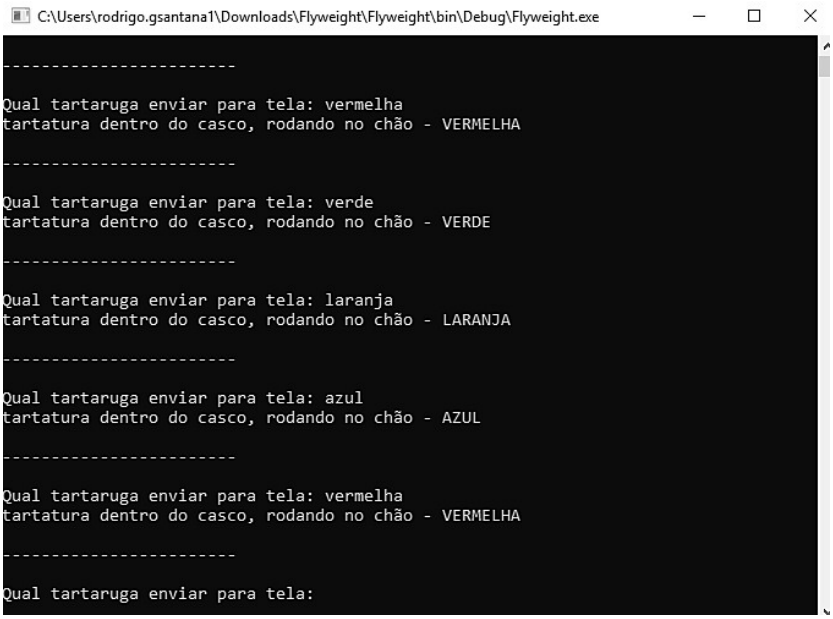

C:\Users\rodrigo.gsantana1\Downloads\Flyweight\Flyweight\bin\Debug\Flyweight.exe
Qual tartaruga enviar para tela: vermelha
tartatura dentro do casco, rodando no chão - VERMELHA
Qual tartaruga enviar para tela: verde
tartatura dentro do casco, rodando no chão - VERDE
Qual tartaruga enviar para tela: laranja
tartatura dentro do casco, rodando no chão - LARANJA
Qual tartaruga enviar para tela: azul
tartatura dentro do casco, rodando no chão - AZUL
Qual tartaruga enviar para tela: vermelha
tartatura dentro do casco, rodando no chão - VERMELHA
Qual tartaruga enviar para tela:

Figura 6.4: Resultado do Pattern Flyweight

Logo, ao solicitar três tartarugas vermelhas, uma verde, uma laranja e uma azul tem-se no total seis tartarugas na tela. Contudo, quero lhe mostrar que apenas quatro tartarugas foram instanciadas. Para tanto, vou colocar um `Break Point` (ponto de interrupção) no `return` do método `GetTartaruga` da classe `Flyweight` para que o programa pare nesta linha e permita visualizar o conteúdo da `lista_de_tartarugas` , inspecionando o objeto. Veja na imagem a seguir o resultado.

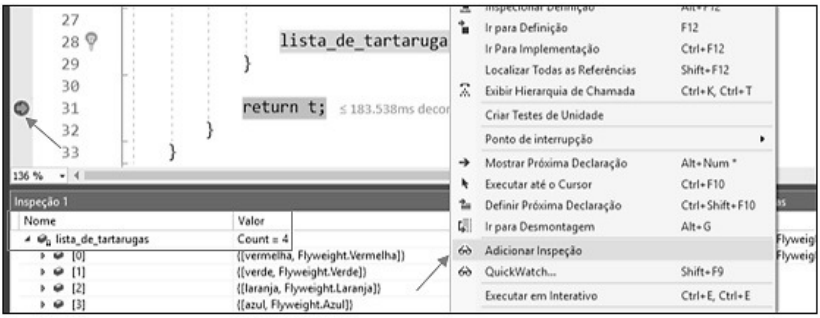


Figura 6.5: Inspecionando nossa lista de instancias

Conforme é possível visualizar, conseguimos solucionar e concluir o desafio. Pois, há seis tartarugas na tela, mas apenas quatro instâncias, ou seja, apenas uma única instância da tartaruga vermelha foi realizada para as três tartarugas vermelhas solicitadas. Muito legal, não é mesmo?

E para finalizar, você lembra da tradução de Flyweight que é "peso-mosca"? Será que de fato, faz sentido esse significado? Pense então que, agora, poderíamos ter mil tartarugas vermelhas na tela, e mesmo assim o jogo não ficaria sobrecarregado, pois uma única instância seria alocada na memória, podendo parafrasear que "essas mil tartarugas seriam tão leves quanto o peso de uma

mosca". Vejo você no próximo padrão de projeto.

> **Lembrete**
>
> Você pode conferir o código-fonte completo do Design Pattern Flyweight que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:
>
> https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp
>
> Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Flyweight**.

6.3 BRIDGE

Passaremos agora a explorar o padrão Bridge, um Design Pattern classificado com o propósito estrutural e de escopo objeto. A grande estratégia desse padrão é dividir uma classe ou um conjunto de classes relacionadas em duas hierarquias separadas: uma responsável pela representação (abstração) do objeto que deverá ser criado e a outra responsável pela implementação. Ainda, este padrão permite o desenvolvimento dos objetos que compõem essa hierarquia de maneira independente umas das outras. Vamos enfrentar mais um desafio e na prática entender e vislumbrar um exemplo de codificação desse padrão.

Hora do desafio

Tetris é um jogo clássico, no qual formas descem a tela e o objetivo é empilhá-las encaixando umas nas outras. Você já jogou Tetris, certo?

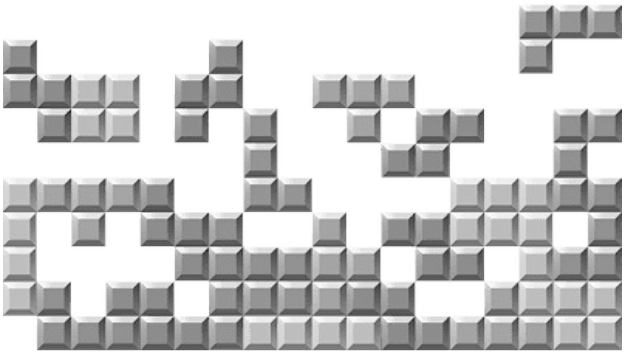


Figura 6.6: Ilustração do jogo Tetris

Pois bem, imagine que teríamos que criar as diversas formas e cores para um jogo como esse. Criar um objeto e cor para cada forma possível não será um caminho produtivo. Vamos imaginar que teríamos que criar duas formas, sendo que cada uma pode ser tanto laranja, rosa ou verde.

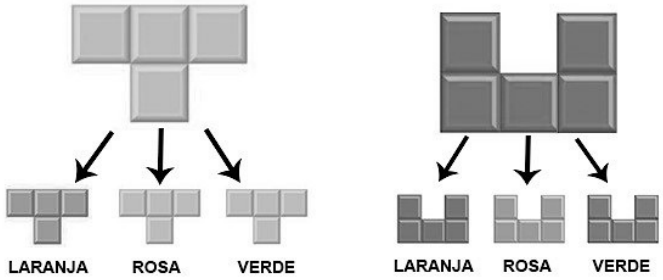


Figura 6.7: Ilustração de formas do jogo Tetris

Conforme é possível visualizar na imagem anterior, teríamos que criar seis objetos, que correspondem à quantidade de possibilidades. Contudo, vamos utilizar o padrão Bridge para reduzir a quantidade de criação de objetos para representar as formas possíveis que terão como referência à cor, uma vez que cada forma pode ter diversas cores. Essa referência funcionará como uma ponte (Bridge) entre as classes `forma` e `cor`. Então vamos estabelecer um vínculo (eis uma ponte) entre forma e cor, para que, sempre que novas cores forem adicionadas, elas automaticamente estarão disponíveis para ser combinadas e/ou utilizadas pelas formas que vão descer pela tela do jogo. Vamos ao código!

Codificando a solução com Bridge

Após criar um projeto `Console` no Visual Studio com o nome Bridge, vamos criar uma interface chamada `IForma` para estabelecer aos objetos que herdarem essa interface que deverão implementar um método `Descer` e um atributo de nome e do

tipo `ICor` que criaremos na sequência. Siga o código a seguir.

```
public interface IForma
{
    string Descer();
    ICor ICor { get; set; }
}
```

Agora, criaremos as formas concretas, herdando e implementando a interface `IForma`. Para tanto, crie duas classes, uma chamada `Forma1` e outra `Forma2`. Ambas vão implementar no método `Descer` um retorno de texto de uma letra que simbolizará uma forma do jogo descendo na tela, concatenando com a cor (que tem como tipo uma interface). Veja na sequência o código-fonte destas duas classes.

```
public class Forma1 : IForma
    {
        public ICor ICor { get; set; }

        public string Descer()
        {
            return "T - " + ICor.Cor();
        }
    }


public class Forma2 : IForma
    {
        public ICor ICor { get; set; }

        public string Descer()
        {
            return "U - " + ICor.Cor();
        }
    }
```

As formas possíveis no jogo em nosso contexto estão criadas, precisamos agora definir as cores possíveis. Para tanto, criaremos uma interface com o atributo `Cor`, veja:

```
public interface ICor
  {
      string Cor();
  }
```

Basta agora implementarmos as cores concretas laranja, rosa e verde. Para cada cor, criaremos uma classe e todas herdam a interface `ICor`. Na implementação do método `Cor`, retornaremos uma `string` com as respectivas cores de acordo com a classe. Confira como fica a codificação dessas classes a seguir:

```
public class Laranja : ICor
{
    public string Cor()
    {
        return "Laranja";
    }
}


public class Rosa : ICor
{
    public string Cor()
    {
        return "Rosa";
    }
}

public class Verde : ICor
{
    public string Cor()
    {
        return "Verde";
    }
}
```

Neste exato momento, criamos duas formas e três cores. Portanto, trabalharemos agora na criação do nosso Bridge que permitirá criar uma ponte entre as formas e as cores que criamos, e

assim, poderemos criar combinações diferentes, por exemplo, uma forma em "U" nas cores laranja e verde, uma outra forma "L" nas cores rosa e verde etc. Logo, na codificação a seguir, você vai notar um método para imprimir na tela qual forma está descendo e um atributo do tipo `IForma`, que é a essência da nossa ponte fazendo a conexão intermediária com a forma solicitada.

```
public class Bridge
    {
        public IForma forma_solicitada { get; set; }

        public void ExibeQualFormaEstaDescendoNaTela()
        {
            Console.WriteLine(forma_solicitada.Descer());

        }
    }
```

Voltemos agora à classe `Programa`, no método `Main`, para utilizarmos de fato o Bridge e colocá-lo para funcionar. Veja a codificação dessa classe logo a seguir.

```
class Program
    {
        static void Main(string[] args)
        {
            Bridge bridge = new Bridge();
            Random random = new Random();

            void Sortear()
            {
                if (random.Next(2) == 1)
                    bridge.forma_solicitada = new Forma1();
                else
                    bridge.forma_solicitada = new Forma2();


                if (random.Next(1,3) == 1)
                    bridge.forma_solicitada.ICor = new Verde();
                else if (random.Next(1,3) == 2)
                    bridge.forma_solicitada.ICor = new Laranja();
```

```
                else
                    bridge.forma_solicitada.ICor = new Rosa();
            }

            Console.WriteLine("Pressione ENTER para enviar uma fo
rma");

            while (1 > 0)
            {
                ConsoleKeyInfo input = Console.ReadKey();
                if (input.KeyChar == 13)
                    Sortear();


                Console.WriteLine();
                bridge.ExibeQualFormaEstaDescendoNaTela();
            }

        }
    }
```

No método principal, primeiro, instanciamos a `bridge` que faz a função de ponte com o objeto solicitado. Perceba que não estamos instanciando nenhuma das formas diretamente; pelo contrário, estamos solicitando para a `bridge` e ela é quem instancia o objeto. Veja a relação entre as classes na imagem:

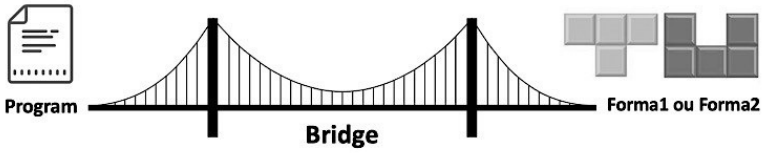


Figura 6.8: Ilustração do Bridge

Na sequência, criaremos um objeto randômico para gerar

números aleatórios com base em um intervalo predefinido. A ideia é sortear o objeto e a cor que deverá descer na tela bem como é no jogo que tomamos como exemplo. Assim sendo, criamos dentro de uma função chamada `Sortear` uma estrutura `If` para instanciar a forma sua cor de acordo com o número sorteado.

Por fim, dentro de laço de repetição que tem uma condição que será sempre verdadeira (um é maior do que zero) definimos que toda vez que a tecla `Enter` for pressionada uma nova forma deve ser gerada, imprimindo na tela qual sua forma e cor.

Agora, vamos pensar na possibilidade de o jogo expandir para 7 cores. Neste caso, adicionaríamos as novas cores e teríamos 9 objetos criados e 14 possibilidades (número de cores multiplicado por formas). Desta forma, perceba que conseguimos dividir um conjunto de classes relacionadas em hierarquias separadas (forma e cor), e ainda, deixar essa hierarquia independente umas das outras, permitindo a qualquer momento adicionar novas cores ou formas. Esse é o Design Pattern Bridge e você conseguiu cumprir mais um desafio, parabéns!

LEMBRETE

Você pode conferir o código-fonte completo do Design Pattern Bridge que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Bridge**.

6.4 COMPOSITE

Quando você ouvir sobre o Design Pattern Composite, pense em uma árvore, pois é exatamente essa a representação hierárquica que esse padrão estabelece, uma vez que, bem como uma árvore, é possível trabalhar com objetos individuais (nós) ou grupo de objetos (galhos). Vamos a mais um desafio, agora, utilizando o Composite.

Hora do desafio

O clássico Super Mario World é composto de várias fases e mapas diferentes. Cada fase segue o tema do mapa, como fases aquáticas, floresta, caverna etc. A cada fase que o jogador conclui, novas fases são liberadas no mapa para que possamos continuar desbravando o jogo até conseguirmos zerá-lo.

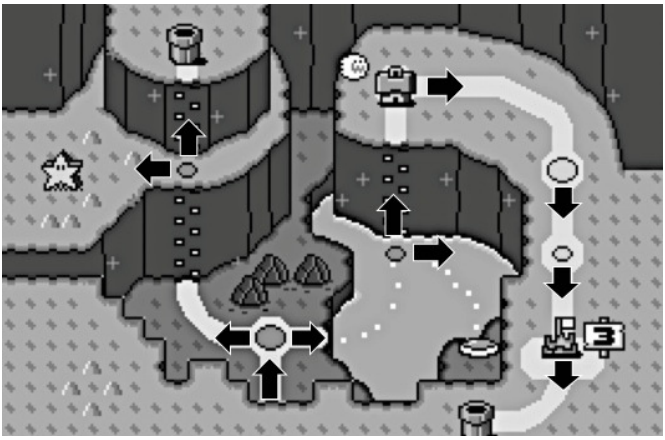


Figura 6.9: Ilustração de fases de um mapa do jogo Super Mario World

Você consegue visualizar algo em comum nessa estrutura de fases do jogo? Pois é, essa é uma hierarquia em árvore, onde existem fases que dependem de outras (nós), sendo que cada conjunto de fases faz parte de um ramo de nós (galho). Veja na imagem a seguir uma ilustração de exemplo.

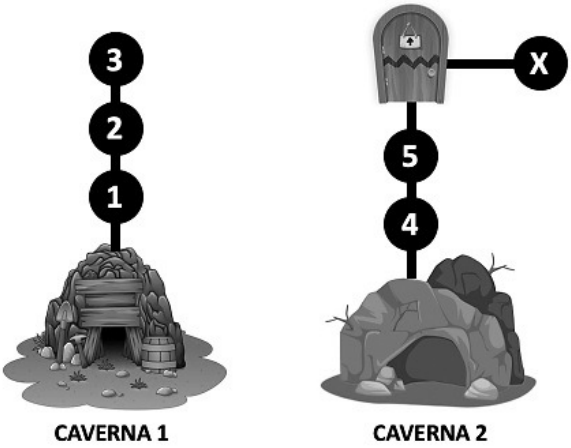


Figura 6.10: Ilustração de hierarquia entre fases de um jogo de exemplo

Logo, o nosso desafio será criar essa estrutura com os objetos. Tudo pronto?

Codificando a solução com Composite

Primeiro, vamos criar uma classe abstrata que será herdada pelas fases do nosso jogo, que terão nome e métodos para permitir adicionar, remover e mostrar. Veja o código-fonte:

```
    public abstract class ComponenteFase
    {
        protected string nome;

        public ComponenteFase(string nome)
        {
            this.nome = nome;
        }

        public abstract void Adicionar(ComponenteFase c);
        public abstract void Remover(ComponenteFase c);
        public abstract void Mostrar(int profundidade);
       }
    }
```

Agora, podemos criar o objeto concreto que implementará a representação da fase do jogo que criamos anteriormente. Chamaremos essa classe de `FaseJogo` e o seu código-fonte será o seguinte:

```
public class FaseJogo : ComponenteFase
    {
        public FaseJogo(string nome) : base(nome)
        {

        }

        public override void Adicionar(ComponenteFase c)
        {
            Console.WriteLine("Não é possível adicionar a fase no
 jogo por aqui!");
```

```
        }

        public override void Mostrar(int profundidade)
        {
            Console.WriteLine(new string('-', profundidade) + nom
e);
        }

        public override void Remover(ComponenteFase c)
        {
            Console.WriteLine("Não é possível remover a fase do j
ogo por aqui!");
        }
    }
```

Perceba que nessa classe, no método construtor, utilizamos a palavra reservada `base` solicitando o parâmetro `nome`. Por meio do comando `base`, é possível acessar membros da classe derivada que nós vamos utilizar no método `Mostrar` para resgatar o nome da caverna (classe derivada) e a qual caverna a subfase pertence, para reproduzirmos a hierarquia das fases na tela. Já os métodos `Adicionar` e `Remover` quando executados, exibirão na tela que não é possível adicionar ou remover fases do jogo diretamente no objeto `FaseJogo`, pois essa será uma responsabilidade do objeto `Composite` que criaremos agora com o seguinte código-fonte:

```
public class Composite : ComponenteFase
    {

        private List<ComponenteFase> fasesjogo = new List<Compone
nteFase>();

        public Composite(string nome): base(nome)
        {

        }

        public override void Adicionar(ComponenteFase c)
        {
            this.fasesjogo.Add(c);
```

```
        }

        public override void Mostrar(int profundidade)
        {
            Console.WriteLine(new string('-', profundidade) + nom
e);

            foreach (ComponenteFase item in this.fasesjogo)
            {
                item.Mostrar(profundidade + 2);
            }
        }

        public override void Remover(ComponenteFase c)
        {
            this.fasesjogo.Remove(c);
        }
    }
```

No objeto `Composite`, criamos uma lista que vai armazenar as fases do jogo. No método `Adicionar` é possível adicionar uma nova fase, tal como excluir, no método `Remover`. Já o método `Mostrar` recebe como parâmetro a `profundidade`, que vamos utilizar para incrementar um traço na tela para facilitar a visualização de qual fase está dentro de qual caverna. Ainda, neste método, em um laço de repetição, percorre-se a lista de fases do jogo para exibir na tela.

Finalmente, podemos ir ao método principal da classe `Program`. No método `Main`, vamos iniciar criando um objeto `mapa` instanciando o `Composite`, que será o mapa do nosso jogo. Em seguida, criaremos dois novos objetos também instanciando `Composite`, que serão nossa primeira e segunda caverna do jogo, adicionando a cada uma delas suas respectivas subfases. Na sequência, vamos criar uma porta secreta dentro da segunda caverna com uma subfase secreta. Logo, adicionamos no mapa do jogo as cavernas um e dois, e adicionamos a porta secreta

na segunda caverna. Finalmente, solicitamos a exibição das fases do jogo na tela. Veja como fica essa codificação a seguir.

```
class Program
    {
        static void Main(string[] args)
        {
            Composite mapa = new Composite("MAPA DAS CAVERNAS");

            Composite caverna1 = new Composite("Caverna 1");
            caverna1.Adicionar(new FaseJogo("Sub Fase 1"));
            caverna1.Adicionar(new FaseJogo("Sub Fase 2"));
            caverna1.Adicionar(new FaseJogo("Sub Fase 3"));

            Composite caverna2 = new Composite("Caverna 2");
            caverna2.Adicionar(new FaseJogo("Sub Fase 4"));
            caverna2.Adicionar(new FaseJogo("Sub Fase 5"));

            Composite porta_secreta = new Composite("Porta Secret
a");
            porta_secreta.Adicionar(new FaseJogo("Sub Fase Secret
a X"));

            mapa.Adicionar(caverna1);
            mapa.Adicionar(caverna2);
            caverna2.Adicionar(porta_secreta);

            mapa.Mostrar(1);

            Console.ReadKey();

        }
    }
```

Logo, ao executar a nossa solução, teremos o resultado adiante:

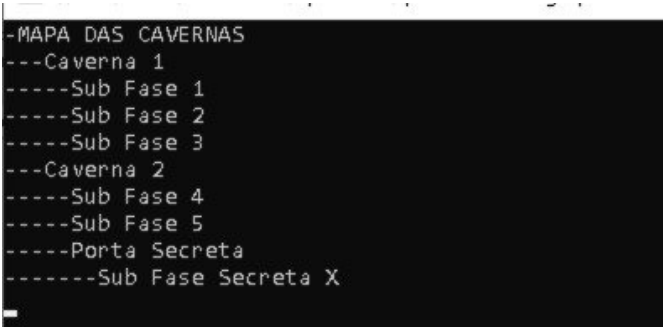


Figura 6.11: Resultado do Pattern Composite

E assim, utilizando o Composite, conseguimos criar uma estrutura em árvore com objetos que são as fases do nosso jogo com fases dentro de fases (nós) e conjuntos de fases (cavernas). Ufa, mais um desafio completo, mas tem muito mais. Até a próxima!

LEMBRETE

Você pode conferir o código-fonte completo do Design Pattern Composite que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Composite**.

6.5 DECORATOR

O padrão Decorator tem como escopo objetos. Por meio dele, é possível anexar novos comportamentos a objetos dinamicamente. Pense que um objeto pode ter um comportamento padrão, mas poderá receber novas responsabilidades dinamicamente ao longo da aplicação. Sendo assim, vamos a mais um desafio e programar um exemplo usando esse Design Pattern.

Hora do desafio

Tomaremos como exemplo um jogo de batalhas épico em que um personagem inicia com uma armadura padrão, mas ao longo do qual poderá adicionar novos objetos à sua armadura, expandindo o comportamento do personagem. Em vista disso, o nosso desafio será possibilitar anexar novos itens à armadura. Vamos lá!

Codificando a solução com Decorator

Primeiramente, vamos criar uma classe abstrata para representar o molde da armadura. Nessa classe, teremos apenas um atributo para armazenar a descrição da armadura, que iniciará com um valor padrão. Contudo, esse atributo terá apenas o método `get` (para resgatar o conteúdo) e utilizaremos a palavra-chave `virtual` para permitir que o atributo seja substituído em uma classe derivada [22].

Acessador get e membros virtuais

O `get` define um meio de acesso que permite acessar e resgatar o conteúdo de uma propriedade. Por outro lado, o `virtual` é um modificador, que, quando definido em um atributo também conhecido como membro, permitirá modificá-lo ou substituí-lo.

```
    public abstract class MoldeArmadura
    {
        private string _descricao = "Armadura do Personagem Abstr
ata";

        public virtual string Descricao
        {
            get { return _descricao; }
        }
    }
```

Agora, criaremos a armadura padrão que vai implementar o molde que acabamos de criar sobrescrevendo a descrição apenas. Veja o código a seguir:

```
public class ArmaduraPadrao : MoldeArmadura
    {
        string _descricao = "Proteção Simples, ";

        public override string Descricao
        {
            get
            {
                return _descricao;
            }
        }
    }
```

Temos o molde e a armadura padrão criada. Agora, criaremos a classe para implementarmos o padrão Decorator para anexar novos elementos. Sendo assim, criaremos uma classe chamada `DecoratorArmadura` que implementa a classe abstrata `MoldeArmadura`.

```
public class DecoratorArmadura : MoldeArmadura
    {
        string _descricao = "Decorador Abstrato da Armadura do Pe
rsoangem";

        public override string Descricao
        {
            get { return _descricao; }
        }
    }
```

Uma vez criada a classe `DecoratorArmadura`, vamos criar as classes `Capacete` e `Espada`, que serão os itens concretos disponíveis para adicionar na armadura padrão implementando a classe `DecoratorArmadura`. Veja como fica a criação dessas duas classes:

```
public class Capacete : DecoratorArmadura
    {
        string _descricao = "Capacte, ";
        MoldeArmadura _moldeArmadura;

        public Capacete(MoldeArmadura moldeArmadura)
        {
            _moldeArmadura = moldeArmadura;
        }

        public override string Descricao
        {
            get
            {
                return _moldeArmadura.Descricao + _descricao;
            }
        }
```

```
    }

public class Espada : DecoratorArmadura
    {
        string _descricao = "Espada Ultra Forte, ";
        MoldeArmadura _moldeArmadura;

        public Espada(MoldeArmadura moldeArmadura)
        {
            _moldeArmadura = moldeArmadura;
        }

        public override string Descricao
        {
            get
            {
                return _moldeArmadura.Descricao + _descricao;
            }
        }
    }
```

Note que os itens concretos (`Capacete` e `Espada`) possuem uma implementação semelhante, uma vez que implementam a mesma classe. Em seus métodos construtores, as classes possuem como parâmetro o molde da armadura e um método `Descricao` que retorna a armadura com o novo objeto que será anexado.

Finalmente, podemos partir para a codificação do método principal do nosso programa. Veja como fica a classe `Program` a seguir.

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(" ### Veste Armadura Padrão ###");
            MoldeArmadura armadura = new ArmaduraPadrao();
            Console.WriteLine("Descrição: " + armadura.Descricao.
TrimEnd(' ', ','));

            Console.WriteLine();
```

```
            Console.WriteLine(" ### Incluir Novos Itens na Armadu
ra (Decorar) ###");
            armadura = new Capacete(armadura);
            armadura = new Espada(armadura);

            Console.WriteLine("Descrição: " + armadura.Descricao.
TrimEnd(' ', ','));

            Console.ReadKey();
        }
    }
```

Em nosso método principal, é instanciada uma armadura padrão no objeto chamado `armadura`, que é do tipo `MoldeArmadura` e que, em seguida, escreve na tela a descrição da armadura padrão. Após saltar uma tela, adicionamos a armadura de dois novos objetos (capacete e espada) e então solicitamos novamente a escrita na tela de descrição, e o resultado é a armadura padrão, mas agora, com dois novos objetos anexados.

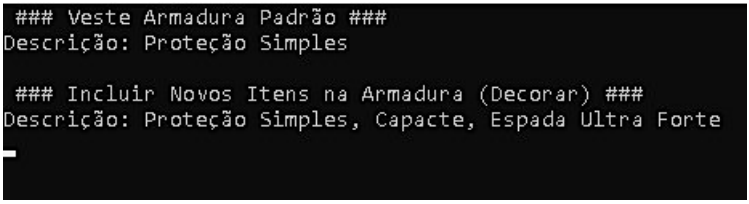

Veste Armadura Padrão
Descrição: Proteção Simples
Incluir Novos Itens na Armadura (Decorar)
Descrição: Proteção Simples, Capacte, Espada Ultra Forte

Figura 6.12: Resultado do Pattern Decorator

Logo, ao inspecionar o objeto `armadura` é possível visualizar que os demais objetos estão anexados.

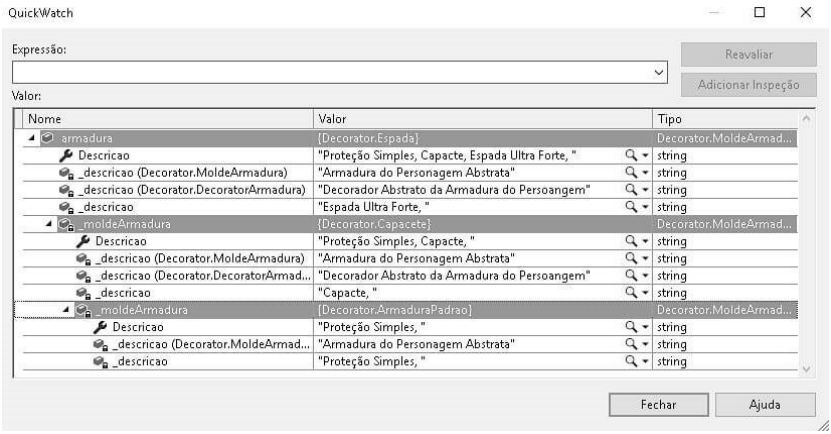


Figura 6.13: Inspecionando o objeto armadura

Desta forma, mais um desafio foi completado. O Design Pattern Decorator permite anexar novos comportamentos a objetos dinamicamente, então, se o jogo precisar de novos itens para armadura, como um escudo, basta adicionar uma nova classe seguindo a mesma lógica de implementação do capacete e espada. Tente você agora criar esse novo objeto, execute e teste. E espero por você no próximo Pattern!

Lembrete

Você pode conferir o código-fonte completo do Design Pattern Decorator que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Decorator**.

6.6 FACADE

O Design Pattern Facade tem como princípio ocultar a complexidade de uma ou mais classes, permitindo isolar partes do sistema (subsistema). Pensando estruturalmente, podemos criar uma "fachada" (Facade, em inglês) que contém alguns subsistemas que poderão ser acessados somente por meio da fachada. Desta forma, a estrutura interna desses subsistemas fica oculta, uma vez que é muito mais simples acessá-los por meio da fachada. Posta essa introdução a esse padrão, vamos a um novo desafio para colocá-lo em prática.

Hora do desafio

Neste desafio, teremos como cenário um jogo épico do estilo RTS, do inglês Real-time strategy ou, em português, jogo de

estratégia em tempo real. Neste jogo, há diversos tipos de tropas e o objetivo é se fortalecer para vencer as tropas inimigas.

Cada tropa tem uma construção fundamental que é o centro de comando. Internamente, no centro de comando, são executadas diversas operações como coletar de recursos de energia, produzir armamentos, treinar guerreiros etc. Não nos importa saber como cada operação (subsistema) do centro de controle é realizada internamente, queremos que o centro de controle apenas as mande fazer. Diante disso, esse será o nosso desafio: criar uma estrutura que oculte como as operações internas são executadas, permitindo que o centro de controle apenas mande executá-las.

Codificando a solução com Facade

Primeiro, vamos criar os subsistemas que o centro de controle (nossa base) poderá executar. Em nosso exemplo, vamos criar três classes, cada uma para representar um subsistema. Veja, a seguir, o código-fonte dessas três classes.

```
public class SubSistemaUm
    {
        public void Responsabilidade()
        {
            Console.WriteLine("Coletar Recursos de Energia para a
 base");
        }
    }

    public class SubSistemaDois
    {
        public void Responsabilidade()
        {
            Console.WriteLine("Produzir Armamento para Guereiros"
);
        }
    }
```

```
public class SubSistemaTres
{
    public void Responsabilidade()
    {
        Console.WriteLine("Treinar Guerreiros");
    }
}
```

As três classes possuem a mesma estrutura contando com um método chamado `Responsabilidade` que escreve na tela o que executará cada subsistema. Agora, criaremos a classe `Facade` que representará o nosso centro de controle. Veja como fica essa classe:

```
public class Facade
    {
        private SubSistemaUm um;
        private SubSistemaDois dois;
        private SubSistemaTres tres;

        public Facade()
        {
            this.um = new SubSistemaUm();
            this.dois = new SubSistemaDois();
            this.tres = new SubSistemaTres();
        }

        public void OperacaoA()
        {
            Console.WriteLine("\nOperação A ------- ");
            this.um.Responsabilidade();
            this.dois.Responsabilidade();
        }

        public void OperacaoB()
        {
            Console.WriteLine("\nOperação B ------- ");

            this.tres.Responsabilidade();
        }
    }
```

Na classe `Facade`, iniciamos com a declaração privada dos

subsistemas que serão de responsabilidade de execução interna da classe, e no método construtor instanciamos esses subsistemas. Ainda, essa classe conta com dois métodos (`OperacaoA` e `OperacaoB`) e dentro de cada operação as responsabilidades dos subsistemas são invocadas. Note que essa classe é quem vai solicitar para as responsabilidades dos subsistemas que sejam executadas.

Por fim, no método principal, vamos usar o Pattern desenvolvido instanciando a classe `Facade` . E então, vamos solicitar a `facade` para executar as operações (`OperacaoA` e `OperacaoB`). Veja o código a seguir.

```
class Program
    {
        static void Main(string[] args)
        {
            Facade facade = new Facade();
            facade.OperacaoA();
            facade.OperacaoB();

            Console.ReadKey();
        }
    }
```

Uma vez pronta a nossa solução, vamos executá-la.

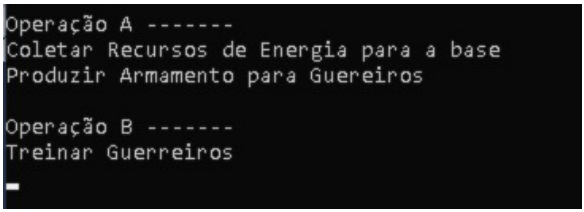


Figura 6.14: Resultado do Pattern Facade

Veja que solicitamos ao `facade` para executar duas operações

(A e B) e cada operação tem um conjunto de subsistemas. A `OperacaoA` tem os subsistemas um e dois e a `OperacaoB` tem o subsistema três. Logo, não foi preciso executar os subsistemas diretamente, pois o `facade` se encarregou disso, e ainda, nem sequer precisamos nos preocupar com a forma como esses subsistemas foram implementados. Missão completa e mais um desafio concluído!

> **Lembrete**
>
> Você pode conferir o código-fonte completo do Design Pattern Facade que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:
>
> https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp
>
> Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Facade**.

6.7 PROXY

O Design Pattern Proxy é um padrão estrutural, que tem como finalidade controlar o acesso a objetos. Ou seja, se um determinado objeto estiver sob controle do Proxy, as solicitações a ele devem passar pelo Proxy, que por sua vez decidirá a execução de algo desejado do objeto.

Convido você a mais um desafio, agora, colocando em ação o

Design Pattern Proxy!

Hora do desafio

Alguns dos jogos clássicos da era de ouro dos videogames contavam com os *passwords* (em português, senhas) que permitiam continuar o jogo de onde havia parado, bastando anotar o código para retomar a fase mais tarde. Logo, o nosso desafio será criar um Proxy para controlar o acesso a uma determinada fase do jogo, que poderá ser jogada somente se o password correto for informado. Vamos ao código!

Hora do desafio

Em nossa solução, vamos começar com a criação de uma simples interface que vai representar a fase do jogo com um método `Jogar`. Veja a seguir o código-fonte:

```
public interface IFase
    {
        string Jogar();
    }
```

E agora, criaremos uma outra classe chamada `FaseJogo`. Note que essa classe não vai implementar a interface que criamos anteriormente, pois deixaremos isso para o Proxy que criaremos a seguir. Será o Proxy quem controlará o acesso a essa classe. Veja como fica o código-fonte da classe `FaseJogo`.

```
public class FaseJogo
    {
        public string Jogar()
        {
            return "Você está de volta a fase do jogo!";
        }
    }
```

Partiremos para a criação da classe na qual vamos implementar o Proxy, chamada de `ProxyFase`.

```
public class ProxyFase : IFase
    {
        FaseJogo fasejogo;
        string password = "123";

        public string Jogar()
        {
            if (this.fasejogo != null)
                return fasejogo.Jogar();

            return "Informe o PASSWORD correto para abrir a fase
do jogo!";
        }

        public string InformarPassword(string codigo)
        {
            if(codigo == this.password)
            {
                this.fasejogo = new FaseJogo();
                return "Password Correto!";
            }

            return "Password Incorreto!";
        }
    }
```

Em nossa classe `ProxyFase` implementamos a interface `IFase` e aqui, no método `Jogar`, por meio de uma estrutura condicional, verificamos se a fase do jogo já está instanciada. Caso já esteja (não nula), é executado o método `jogar`, mas agora, de outra classe chamada `FaseJogo`, instanciada no início da classe que estamos trabalhando; caso contrário (`fasejogo` é nula), então é retornada a mensagem solicitando o password da fase.

Ainda nesta classe, temos o método `InformarPassword` que recebe como parâmetro uma `string` e então verifica se o valor

informado é o password correto para liberar a fase para ser jogada e então instancia a fase. Ufa! Esse é o nosso Proxy. Vamos à classe `Program` e fazer alguns experimentos no método principal do nosso projeto. Siga os passos que codificaremos na sequência.

Primeiro, vamos instanciar diretamente o objeto `FaseJogo` e invocar o método `Jogar` para que você possa ver que desta forma, a classe `FaseJogo` é independente.

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("#### Acessando a Fase do Jogo sem
o Proxy ####");
            FaseJogo fase = new FaseJogo();
            Console.WriteLine(fase.Jogar());
            Console.WriteLine();

            Console.ReadKey();
        }
    }
```

O resultado é a fase liberada para jogar sem ter digitado o password (senha).

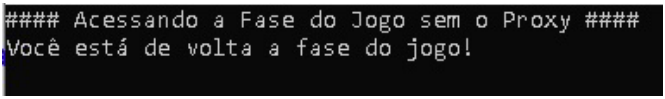


Figura 6.15: Fase liberada para jogar sem ter digitado o password

Contudo, vamos agora, colocar o nosso Proxy na jogada para controlar o acesso à classe `FaseJogo` . Logo, vamos instanciar o `ProxyFase` e invocar o método `Jogar` da mesma maneira que acabamos de fazer, mas com o proxy.

```
class Program
```

```
    {
        static void Main(string[] args)
        {

            Console.WriteLine("#### Usando o Proxy para controlar
 o acesso a fase do jogo ####");
            Console.WriteLine();
            ProxyFase proxy = new ProxyFase();

            Console.WriteLine(proxy.Jogar());

            Console.ReadKey();
        }
    }
```

```
#### Usando o Proxy para controlar o acesso a fase do jogo ####

Informe o PASSWORD correto para abrir a fase do jogo!
```

Figura 6.16: Controlando o acesso com o Proxy

Note que, utilizando o Proxy, foi solicitado o password para liberar a fase do jogo. Vamos informar um password inválido e um válido para testarmos o comportamento do Proxy. Veja como fica o código-fonte do método `Main` a seguir.

```
class Program
    {
        static void Main(string[] args)
        {
            ProxyFase proxy = new ProxyFase();

            Console.WriteLine("#### Tentando acessar a fase do jo
go com Password incorreto ####");
            Console.WriteLine(proxy.InformarPassword("465"));
            Console.WriteLine(proxy.Jogar());

            Console.WriteLine();

            Console.WriteLine("#### Tentando acessar a fase do jo
go com Password correto ####");
```

```
            Console.WriteLine(proxy.InformarPassword("123"));
            Console.WriteLine(proxy.Jogar());
            Console.WriteLine();

            Console.ReadKey();
        }
    }
```

Conforme a ilustração do resultado, na imagem a seguir, veja que na primeira tentativa ao informar o código inválido o Proxy retornou que o password é incorreto. Já na segunda tentativa, agora informando o código correto, além confirmar que o password está correto o Proxy liberou a fase do jogo. Legal, não é mesmo?

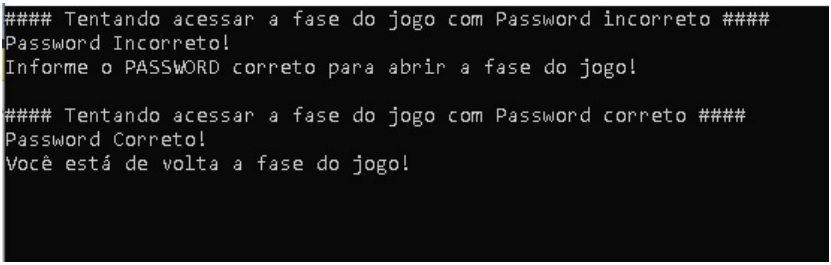

Tentando acessar a fase do jogo com Password incorreto
Password Incorreto!
Informe o PASSWORD correto para abrir a fase do jogo!
Tentando acessar a fase do jogo com Password correto
Password Correto!
Você está de volta a fase do jogo!

Figura 6.17: Resultado do Pattern Proxy

Muito bem, mais um desafio foi concluído com sucesso, pois você utilizou o Design Pattern Proxy para controlar o acesso a um objeto. Assim sendo, esse padrão encerra o conjunto dos Design Pattern estruturais.

Lembrete

Você pode conferir o código-fonte completo do Design Pattern Proxy que aqui desenvolvemos. A solução está disponível em nosso repositório no GitHub no seguinte endereço:

https://github.com/rodrigogoncalvessantana/aprendendo-design-patterns-com-games-csharp

Em seguida, acesse o diretório **Design Patterns de Estrutura** e você verá o diretório **Proxy**.

Conclusão

Neste capitulo, você concluiu a saga dos Design Patterns clássicos de estrutura. Iniciamos com o padrão **Adapter**, que pode ser utilizado tanto para classe ou objetos como um intermediador, tal como um adaptador é um intermediário entre tomadas de energia de padrões de pinos diferentes. Em seguida, vimos o **Flyweight**, que permite compartilhar objetos com características parecidas reaproveitando objetos existentes sem a necessidade de realizar novas instâncias. Na sequência, o padrão da vez foi o **Bridge**, que permite dividir uma classe ou um conjunto de classes relacionadas em hierarquias separadas, de maneira independente umas das outras. Vimos também o padrão **Composite**, que estabelece uma representação hierárquica em árvore trabalhando com objetos individuais (nós) ou grupo de objetos (galhos). Ainda, aprendemos o Design Pattern **Decorator**, que permite anexar

novos comportamentos a objetos dinamicamente. Em seguida, o alvo foi o **Facade** que tem como princípio ocultar a complexidade de uma ou mais classes, permitindo isolar partes do sistema (subsistema). Por fim, encerramos mais um capítulo com muita prática com o Design Pattern **Proxy**, que tem como finalidade controlar o acesso a objetos.

Figura 6.18: A saga dos Design Patterns de estrutura

Logo, você concluiu os Design Patterns clássicos de estrutura, parabéns! No próximo capítulo, vamos desbravar nossa última saga, a dos Design Patterns de comportamento. Vejo você no próximo capítulo.

Referências

[18] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/protected

[19] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/void

[20] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/this

[21] https://docs.microsoft.com/pt-br/dotnet/api/system.collections.generic.dictionary-2?

view=netframework-4.7.2

[22] https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/virtual

Capítulo 7

DESIGN PATTERNS DE COMPORTAMENTO

Neste capítulo será abordado o desenvolvimento dos padrões de **comportamento**, cujo objetivo é definir as relações entre objetos ao definir padrões de comunicação entre si. Este será mais um capítulo bem prático, portanto, abra o seu Visual Studio e vamos em frente!

7.1 TEMPLATE METHOD

Veremos agora um Design Pattern com proposito comportamental, cujo escopo é classe. O padrão Template Method, ou em português, método de modelo, define um modelo de passos abstratos que serão fornecidos às subclasses para que possam implementar os passos de um algoritmo sem alterar a estrutura do próprio algoritmo.

Portanto, ao desenvolver os componentes, devemos decidir quais etapas de um algoritmo são invariantes (ou padrão) e quais são variantes (ou personalizáveis). Os passos que forem determinados como invariantes do algoritmo serão implementados em uma classe base abstrata, enquanto os passos variantes recebem uma implementação padrão ou nenhuma.

Nos passos variantes, pensaremos analogicamente como "ganchos" ou "espaços reservados", que podem ou devem ser fornecidos pela classe derivada concreta. Além do mais, permite que a subclasse estenda ou substitua algum número dessas etapas.

Hora do desafio

Neste desafio, utilizaremos como exemplo um jogo de corrida. Em nosso jogo, será possível escolher a dificuldade das corridas, entre fácil, normal e difícil. Porém, a sequência de pistas e a trilha sonora são as mesmas; desta forma, elas são as invariantes (padrão, pois não mudam). Mas, de acordo com a dificuldade escolhida, teremos características variantes, isto é, diferentes entre um nível e outro.

Convido você a codificar esse desafio utilizando o Design Pattern Template Method. Teremos que definir passos invariantes do jogo independente do modo de dificuldade escolhido pelo jogador (padrão) e definir "espaços reservados" como características variantes do jogo que podem mudar de acordo com o modo de dificuldade selecionado.

Codificando a solução com Template Method

Começaremos codificando a solução do nosso desafio pela classe abstrata `Jogo`. Vamos definir dois métodos variantes que representam o algoritmo de duas fases do jogo (`PrimeiraFase` e `SegundaFase`) que poderão mudar de acordo com a dificuldade. Mas o método `TrilhaSonora` é invariante, pois, independente do nível de dificuldade do jogo, a trilha será a mesma conforme já implementado no próprio método ("Música emocionante").

```
public abstract class Jogo
    {
        public Jogo()
        {
     TrilhaSonora();
            PrimeiraFase();
            SegundaFase();
        }

        public abstract void PrimeiraFase();
        public abstract void SegundaFase();

        private void TrilhaSonora()
        {
            Console.WriteLine("Música emocionante");
        }
    }
```

Observe que todos os métodos são convocados no construtor da classe `Jogo`, o que significa que esse é o algoritmo padrão do jogo em que, depois de definir a trilha sonora, vem a primeira fase, depois, a segunda fase e assim por diante. Veja agora as implementações concretas da classe `Jogo`.

```
public class ModoFacil : Jogo
    {
        public override void PrimeiraFase()
        {
            Console.WriteLine("Combustível para a corrida toda");
        }

        public override void SegundaFase()
        {
            Console.WriteLine("Carros adversários devem correr me
nos");
        }
   }

public class ModoNormal : Jogo
    {
        public override void PrimeiraFase()
        {
```

```
            Console.WriteLine("O carro precisa abastecer uma vez"
);
        }

        public override void SegundaFase()
        {
            Console.WriteLine("Os carros devem correr na mesma ve
locidade");
        }
    }

public class ModoDificil : Jogo
    {
        public override void PrimeiraFase()
        {
            Console.WriteLine("Adicionar obstáculo na pista");
        }

        public override void SegundaFase()
        {
            Console.WriteLine("Carros adversário devem correr mai
s");

        }
    }
```

Criamos três novas classes (`ModoFacil` , `ModoNormal` e `ModoDificil`) e em cada fase nós personalizamos as características de acordo com a dificuldade do jogo.

Agora, deixaremos que o jogador escolha o grau de dificuldade. Para isso, vamos codificar o nosso método principal (`Main`) a seguir.

```
static void Main(string[] args)
        {
            Console.WriteLine("### Escolha o modo de corrida ###"
);
            Console.WriteLine("1-Fácil | 2-Normal | 3-Difícil");

            Console.WriteLine("Suas condições de jogo são: ");
            Jogo jogo = null;
```

```
        switch (Console.ReadLine())
        {
            case "1": jogo = new ModoFacil(); break;
            case "2": jogo = new ModoNormal(); break;
            case "3": jogo = new ModoDificil(); break;
        }

        Console.ReadKey();

    }
```

No método principal, primeiro, escrevemos na tela as opções de jogo que o jogador pode escolher, sendo o número 1 para fácil, 2 para normal e 3 para difícil. Em seguida, capturamos qual foi a opção selecionada pelo jogador e, utilizando uma estrutura *switch case*, instanciamos a classe de acordo com a dificuldade escolhida. Por fim, serão exibidas na tela as condições de jogo. Por exemplo, ao escolher o modo fácil, temos o seguinte resultado conforme a imagem a seguir.

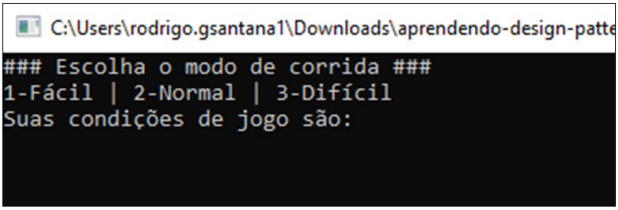

C:\Users\rodrigo.gsantana1\Downloads\aprendendo-design-patte
Escolha o modo de corrida
1-Fácil | 2-Normal | 3-Difícil
Suas condições de jogo são:

Figura 7.1: Exemplo de resultado do Template Method

Você também pode fazer novos testes escolhendo os outros níveis de dificuldade (normal e difícil) e verificar se as características de cada fase mudam. E assim, concluímos o desafio, onde criamos "um modelo" com uma sequência de passos (fases do jogo) abstratos e fornecemos às subclasses (modos de dificuldade) para que cada fase pudesse implementar os passos de

acordo com o nível de dificuldade do jogo.

7.2 INTERPRETER

Eu, particularmente, gosto muito de ir ao cinema e assistir a um bom filme, contudo, nem sempre o filme é dublado (em português). Independente do idioma original do filme, todos podem ler as legendas (tradução) em português, sem precisar conhecer ou compreender o outro idioma, não é mesmo?

Essa é uma boa analogia para representar o **Design Pattern Interpreter**, pois tal como um idioma possui suas regras, ou seja, a gramática. O Pattern Interpreter define uma representação para sua gramática (conjunto de regras) junto com um interpretador que é capaz interpretar a linguagem solicitada. Desde modo, convido você a um desafio com o Interpreter.

Hora do desafio

O cenário do nosso desafio será um jogo de sobrevivência na selva. Em nosso jogo, existem diversas ferramentas e armamentos possíveis de serem escolhidos para jogar, mas precisamos definir uma regra para que o personagem coloque em sua mochila duas ferramentas e um armamento para defender-se dos perigos que ele encontrará. Para tanto, vamos desenvolver uma mochila que será inteligente, como um robô que será capaz de dizer o que está disponível dentro ela rapidamente quando o jogador solicitar.

Codificando a solução com Interpreter

Começaremos a desenvolver nossa solução pela interface que

vai representar as expressões interpretadas pelo robô responsável por montar a mochila do nosso aventureiro, contando com um método que chamaremos de `Interpretar` .

```
public interface IExpressao
    {
        void Interpretar(Contexto contexto);
    }
```

Note que o método possui um parâmetro do tipo `Contexto` que será uma classe que criaremos apenas para armazenar o conteúdo interpretado.

```
    public class Contexto
    {
        public string Conteudo { get; set; }
    }
```

Em seguida, criaremos uma interface simples para representar as ferramentas disponíveis no jogo. E, logo na sequência, criaremos como ferramentas binóculos, bússola e corda, implementando a interface `IFerramenta` . Vamos criar esse código a seguir.

```
public interface IFerramenta : IExpressao
    {
    }

public class Binoculos : IFerramenta
    {
        public void Interpretar(Contexto contexto)
        {
            contexto.Conteudo += string.Format(" {0} ", " Binócul
os ");
        }
    }

    public class Bussola : IFerramenta
    {
        public void Interpretar(Contexto contexto)
        {
            contexto.Conteudo += string.Format(" {0} ", " Bússola
```

```
 ");
        }
    }

    public class Corda : IFerramenta
    {
        public void Interpretar(Contexto contexto)
        {
            contexto.Conteudo += string.Format(" {0} ", " Corda "
);
        }
    }
```

Da mesma forma, faremos a opção de armamento no jogo, criando uma interface chamada `IArmamento`. Na sequência, crie a classe `ArcoFlecha`. Deixo com você e sua criatividade a opção de criar mais opções, todas implementando a interface `IArmamento`.

```
public interface IArmamento : IExpressao
    {
    }
    public class ArcoFlecha : IArmamento
    {
        public void Interpretar(Contexto contexto)
        {
            contexto.Conteudo += string.Format(" {0} ", " Arco e
Flecha ");
        }
    }
```

Neste momento, temos opções de ferramentas e armamento. Agora estamos prontos para a criação da mochila.

```
public class Mochila : IExpressao
    {
        private readonly IFerramenta ferramenta_principal;
        private readonly IFerramenta ferramenta_secundaria;
        private readonly IArmamento armamento;

        public Mochila(IFerramenta ferramenta_principal, IFerrame
nta ferramenta_secundaria, IArmamento armamento)
        {
```

```
            this.ferramenta_principal = ferramenta_principal;
            this.ferramenta_secundaria = ferramenta_secundaria;
            this.armamento = armamento;
        }

        public void Interpretar(Contexto contexto)
        {
            contexto.Conteudo += "Abrindo mochila... \n";

            armamento.Interpretar(contexto);
            contexto.Conteudo += "- 1º Ferramenta";
            ferramenta_principal.Interpretar(contexto);
            contexto.Conteudo += "- 2º Ferramenta";
            ferramenta_secundaria.Interpretar(contexto);

            contexto.Conteudo += "\n... Fechando mochila";

            Console.WriteLine(contexto.Conteudo);
        }
    }
```

A classe `Mochila` inicia com três atributos privados somente leitura que recebem seus respectivos conteúdos somente por meio do método construtor dessa classe. Essa classe implementa o método `Interpretar`, que tem como parâmetro `Contexto`, que armazena e escreve na tela o significado (tradução) do que foi colocado dentro da mochila.

Finalmente, na classe `Programa`, vamos escolher o que colocar dentro da mochila do nosso aventureiro.

```
    class Program
    {
        static void Main(string[] args)
        {
            Mochila mochila = new Mochila(new Corda(), new Binocu
los(), new ArcoFlecha());
            mochila.Interpretar(new Contexto());
            Console.ReadKey();
        }
    }
```

Note que instanciamos a `mochila` e passamos como parâmetro alguns objetos, mas não dizemos nada além disso. Em seguida, solicitamos o conteúdo da mochila (interpretar), o que vai nos retornar o nome dos objetos organizados pelo armamento, ferramenta primária e secundária, além de exibir uma mensagem de que a mochila foi aberta e fechada.

```
Abrindo mochila...
  Arco e Flecha  - 1º Ferramenta  Corda  - 2º Ferramenta  Binóculos
... Fechando mochila
```

Figura 7.2: Resultado do Interpreter

Sendo assim, você desenvolveu e utilizou um padrão que avalia (ferramentas e armamento escolhido) e interpreta as instruções escritas em uma linguagem (mostra o conteúdo interpretado da mochila). Fica para você, como desafio, testar outras possibilidades, como escolher a ferramenta corda, por exemplo, e executar a aplicação para conferir o resultado.

7.3 OBSERVER

O Design Pattern Observer (observador) é um padrão de projeto comportamental, cujo escopo é objeto. Esse padrão permite que objetos sejam informados sobre mudanças ocorridas em outros objetos. Interessante, não é mesmo?

Para facilitar o entendimento desse padrão, pensaremos em um contexto em que poderemos entender melhor sua proposta. Logo, convido você a um novo desafio!

Hora do desafio

Neste desafio, gostaria de lhe perguntar se você já jogou o glorioso Final Fight? Esse jogo é um clássico lançado em 1989 da categoria de games chamada "Beat 'em ups" (conhecido também como *brawlers* ou briga de rua, em português).

Figura 7.3: Ilustração do Jogo Final Fight

Conforme ilustrado na imagem anterior, na parte superior (indicado pela seta na imagem), mostra-se por meio de uma barra vertical a quantidade de "vida" do personagem escolhido, que nosso caso é o Cody. A cada golpe que Cody leva dos outros adversários, a barra vai preenchendo-se da cor vermelha e o personagem perde quando a barra toda estiver vermelha.

Na cena que a imagem está ilustrando, são três adversários contra o personagem Cody, sendo assim, o nosso desafio é criar um algoritmo que avise os três adversários sempre quando o Cody for acertado com um golpe, isto é, quando seu ponto em vida no

jogo diminuir. Para resolver esse desafio, vamos utilizar o **Design Pattern Observer**. Vamos desenvolver um "observador" que comunicará os três adversários sempre quando o Cody for acertado. Vamos lá?

Codificando a solução com Observer

Para solucionar o nosso desafio de "avisar" os três inimigos toda vez que o personagem Cody for acertado, iniciaremos com a criação da interface para representar os personagens do jogo. Veja a seguir o código-fonte.

```
public interface IPersonagem
    {
        void RegistrarObservador(IObservador observador);
        void NotificarPersonagens();
    }
```

Criaremos também outra interface, mas agora, para representar o comportamento do observador, conforme o código a seguir.

```
   public interface IObservador
    {
        void Avisar(IPersonagem personagem);
    }
```

Uma vez que temos uma interface para representar os personagens do jogo e outra para observar as alterações, podemos então partir para a implementação das classes concretas. Iniciaremos pela implementação do personagem Cody, veja logo adiante.

```
public class Cody : IPersonagem
    {
        private List<IObservador> inimigos = new List<IObservador
>();
```

```
        public int vida = 100;

        public void NotificarPersonagens()
        {
            foreach (IObservador i in inimigos)
            {
                i.Avisar(this);
            }
        }

        public void RegistrarObservador(IObservador i)
        {
            inimigos.Add(i);
        }

        public void Golpe_Acertado(bool golpe)
        {
            if (golpe)
                vida -= 10;

            NotificarPersonagens();
        }

        public int getVida()
        {
            return vida;
        }
    }
```

Primeiro, criamos uma lista usando uma coleção genérica que necessita ser referenciada no código com `using System.Collections.Generic;` . Essa lista, que chamamos de `inimigos` , é do tipo `IObservador` (nossa interface) e é utilizada no método `NotificarPersonagens` . Ela contém os inimigos que deverão ser avisados toda vez que o personagem Cody for acertado com um golpe por meio do laço de repetição *foreach*, executando o método `Avisar` para cada inimigo.

Na sequência, temos o método `RegistrarObservador` , que adiciona na lista os `inimigos` que deverão ser avisados toda vez

que o personagem Cody for acertado, que, em nosso exemplo, serão três.

Já o método `Golpe_Acertado` recebe como parâmetro um valor booleano (verdadeiro ou falso) que indica se o personagem Cody foi acertado ou não. Caso o personagem tenha sido acertado, então são descontados de sua vida dez pontos e, em seguida, é executado o método `NotificarPersonagens` . A vida do Cody inicia com o valor 100 (conforme definido no escopo global da classe na variável chamada `vida`). E, por fim, temos o método `getVida` que retorna quantos pontos de vida o personagem Cody ainda tem no jogo.

Agora, partiremos para o desenvolvimento da classe concreta que representará os inimigos em nosso jogo, a classe `Inimigo` , que vai implementar a interface `IObservador` conforme a seguir.

```
class Inimigo : IObservador
    {

        private Cody personagemObservado;

        public void Avisar(IPersonagem personagem)
        {
            if (personagem == personagemObservado)
            {
                Console.WriteLine("o Cody foi acertado com um gol
pe, agora sua vida de jogo é de: " + personagemObservado.getVida(
));
            }
        }

        public Inimigo(Cody cody)
        {
            personagemObservado = cody;
            personagemObservado.RegistrarObservador(this);
        }
    }
```

Nesta classe, criamos o método `Avisar`, que recebe como parâmetro uma interface do tipo `IPersonagem`. Esse método compara se o personagem é o personagem observado, no caso o Cody, conforme definido no escopo global da classe. Se sim, escreve na tela que o personagem Cody foi acertado, concatenando com sua pontuação de vida atual no jogo ao chamar o método `getVida`.

E, por fim, definimos o método `Inimigo`, que recebe como parâmetro um objeto que é do tipo Cody, que tem como objetivo dizer o que personagem observado é o Cody e registrá-lo como sendo observado.

Agora, podemos finalizar nossa solução na classe `Program`, no método `Main`.

```
static void Main(string[] args)
        {
            Cody cody = new Cody();

            Inimigo inimigo1 = new Inimigo(cody);
            Inimigo inimigo2 = new Inimigo (cody);
            Inimigo inimigo3 = new Inimigo (cody);

            while(true)
            {
                Console.WriteLine("Acertar o Cody com um golpe (S
 ou N)?");

                if (Console.ReadLine() == "s")
                    cody.Golpe_Acertado(true);
                else
                    Console.WriteLine("Você não acertou o Cody!")
;
            }
  }
```

Conforme codificado no código anterior, primeiro

instanciamos o personagem `Cody` e, em seguida, três inimigos passando o valor "cody" como parâmetro, que é quem cada inimigo deve observar durante o jogo.

Criamos um laço de repetição *while* que recebe o valor fixo `true`, o que fará com que nosso programa sempre pergunte se o personagem `Cody` deve ser acertado ou não (letra "s" ou "n"). Caso a opção escolhida seja sim (letra "s"), então é chamada a função `Golpe_Acertado`, que descontará dez pontos da vida de `Cody` no jogo e avisará os inimigos de que `Cody` foi acertado e qual é o valor atualizado do ponto de vida do `Cody`. Caso a opção escolhida seja não (letra "n"), então a mensagem escrita na tela é "Você não acertou o Cody!". Veja o resultado na imagem a seguir.

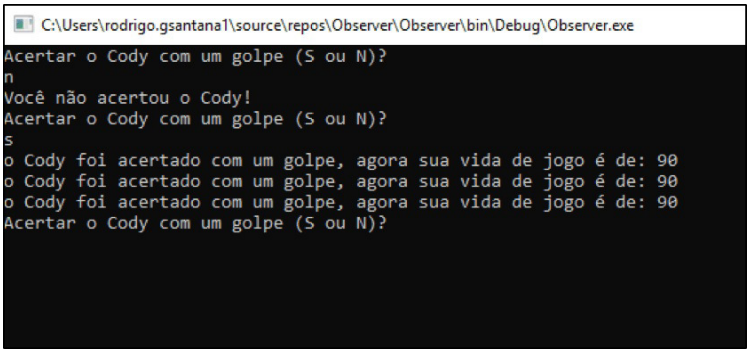

C:\Users\rodrigo.gsantana1\source\repos\Observer\Observer\bin\Debug\Observer.exe
Acertar o Cody com um golpe (S ou N)?
n
Você não acertou o Cody!
Acertar o Cody com um golpe (S ou N)?
s
o Cody foi acertado com um golpe, agora sua vida de jogo é de: 90
o Cody foi acertado com um golpe, agora sua vida de jogo é de: 90
o Cody foi acertado com um golpe, agora sua vida de jogo é de: 90
Acertar o Cody com um golpe (S ou N)?

Figura 7.4: Resultado da solução do jogo desenvolvida com o padrão Observer

Conforme vemos na imagem, na primeira tentativa a opção escolhida foi "não", e então, a mensagem retornada foi de que Cody não foi acertado. Contudo, na segunda tentativa, a opção escolhida de acertar Cody foi "sim". Então, três mensagens aparecem na tela dizendo que Cody foi acertado e que agora sua vida no jogo é de 90 pontos, descontando 10 pontos do valor inicial, que era de 100

pontos. Ou seja, as três mensagens significam que os três personagens inimigos foram avisados de que Cody foi acertado e que, agora, sua vida no jogo e de 90 (noventa) pontos.

Portanto, mais um desafio foi concluído. Com o **Design Pattern Observer** você conseguiu monitorar o comportamento do objeto Cody comunicando todos os personagens inimigos sempre quando o personagem Cody for acertado e informando a quantidade de pontos de vida de Cody no jogo. Parabéns por superar mais esse desafio e chegar até aqui!

7.4 VISITOR

O padrão **Visitor**, ou visitante, permite adicionar novos comportamentos a um objeto existente sem precisar alterar sua estrutura. A ideia é facilitar a adição de novas operações, e quando necessário executar as novas ou operações estendidas é só chamar um Visitor. Vamos a um desafio e colocar isso em prática.

Hora do desafio

Imagine que estamos diante de um jogo que tem suas fases bem definidas, bem como os chefões do jogo. Logo, surge a necessidade de desenvolver uma funcionalidade que exiba a dificuldade da fase ou do chefão, mas não queremos adicionar isso fase a fase, ou chefão por chefão. A solução será utilizar o padrão Visitor, que contará com um método para identificar a dificuldade tanto das fases quanto dos chefões do jogo. Simplesmente chamaremos nosso visitante a contribuir dando essas informações. Vamos ao código!

Codificando a solução com Visitor

Começaremos com a criação da interface que representará o jogo definindo o método `Visitante` .

```
public interface IJogo
    {
        void Visitante(IVisitor visitante);
    }
```

O método `Visitante` recebe como parâmetro a interface `visitante` que vamos criar agora.

```
    public interface IVisitor
    {
        void Identificar(Chefao chefao);
        void Identificar(FaseJogo fasejogo);
    }
```

Note que na interface `IVisitor` definimos dois métodos `Identificar` , pois deixaremos para a aplicação identificar o nível, seja da fase ou do chefão. Sendo assim, vamos à criação das classes que representam jogo e chefão.

```
public class FaseJogo : IJogo
    {
        public string NomeFase { get; set; }

        public void Visitante(IVisitor visitor)
        {
            visitor.Identificar(this);
        }
    }

public class Chefao : IJogo
    {
        public string NomeChefao { get; set; }
        public int PontosVida { get; set; }

        public void Visitante(IVisitor visitante)
        {
```

```
            visitante.Identificar(this);
        }
    }
```

Ambas as classes criadas anteriormente (`FaseJogo` e `Chefao`) implementam a interface `IJogo` . Ainda, no método `Visitante` há o método `Identificar` , que recebe a palavra reservada `this` . Vamos à criação da classe `NivelVisitor` , que conterá a funcionalidade para exibir a dificuldade da fase ou do chefão.

```
public class NivelVisitor : IVisitor
    {
        public void Identificar(FaseJogo fase)
        {
            switch (fase.NomeFase)
            {
                case "Floresta":
                    Console.WriteLine("A fase {0} no jogo é {1}%
difícil", fase.NomeFase, 70);
                    break;
                case "Caverna":
                    Console.WriteLine("A fase {0} no jogo é {1}%
difícil", fase.NomeFase, 30);
                    break;
            }
        }

        public void Identificar(Chefao chefao)
        {
            switch (chefao.NomeChefao)
            {
                case "Boss 1":
                    Console.WriteLine("O chefão {0} é {1}% difici
l e tem {2} pontos de vida.", chefao.NomeChefao, 25, chefao.Ponto
sVida);
                    break;
                case "Boss 2":
                    Console.WriteLine("O chefão {0} é {1}% difici
l e tem {2} pontos de vida.", chefao.NomeChefao, 50, chefao.Ponto
sVida);
                    break;
```

```
            }
        }
    }
```

Note que, agora, é a interface `IVisitor` que está sendo implementada com os respectivos métodos para identificar tanto o nível do jogo quanto do chefão. Vamos então à classe `Programa`, colocar o Visitor em ação.

```
class Program
    {
        static void Main(string[] args)
        {
            List<IJogo> jogo = new List<IJogo>();
            jogo.Add(new FaseJogo() { NomeFase = "Floresta"});
            jogo.Add(new FaseJogo() { NomeFase = "Caverna" });
            jogo.Add(new Chefao() { NomeChefao = "Boss 1", Pontos
Vida = 30 });
            jogo.Add(new Chefao() { NomeChefao = "Boss 2", Pontos
Vida = 50 });

            NivelVisitor niveis = new NivelVisitor();
            foreach (var etapa in jogo)
            {
                etapa.Visitante(niveis);
            }

            Console.ReadLine();
        }
    }
```

No código anterior, da classe `Programa`, para armazenar as fases do jogo e os possíveis chefões criamos uma lista (que faz uso da referência `System.Collections.Generic`), em seguida, adicionamos algumas fases e chefões na lista do jogo. No próximo passo, instanciamos e nomeamos o Visitor de `níveis`, em seguida, por meio de um laço de repetição *foreach* percorremos a lista com o objetivo de mostrar na tela o nível de dificuldade de cada fase e de cada chefão. Veja o resultado a seguir.

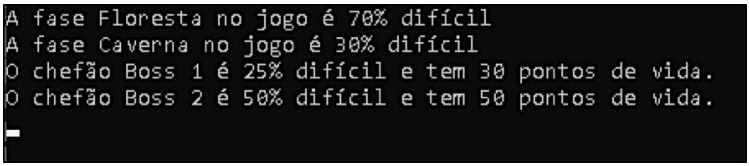

A fase Floresta no jogo é 70% difícil
A fase Caverna no jogo é 30% difícil
O chefão Boss 1 é 25% difícil e tem 30 pontos de vida.
O chefão Boss 2 é 50% difícil e tem 50 pontos de vida.

Figura 7.5: Resultado da solução do jogo desenvolvida com o padrão Visitor

Desta forma, criamos um Visitor que permitiu adicionar novos comportamentos (verificar nível) a um objeto existente (`FaseJogo` e `Chefao`) sem precisar alterar a estrutura dos objetos que já existiam. Parabéns!

7.5 COMMAND

O **Design Pattern Command**, ou comando, em português, permite transformar uma solicitação em um objeto que conterá todas as informações sobre a solicitação. Vamos tomar como exemplo um controle remoto de uma televisão, que normalmente conta com diversas opções, tais como ligar, desligar, aumentar ou diminuir o volume da televisão etc. Logo, o controle remoto é o objeto que tem todas as informações sobre as solicitações permitidas à televisão, sendo apenas um intermediário entre nós e o aparelho.

O interessante desse Pattern é que os objetos não precisam conhecer as operações e/ou comandos solicitados nem o destinatário da solicitação. O controle remoto não precisa saber como as funções são executadas internamente dentro da televisão, ele apenas solicita que as funções sejam executadas. Interessante, não é mesmo? Então, vamos à hora do desafio!

Hora do desafio

Pense que, em um suposto jogo, podemos contar com a ajuda de um robô amigo. Sendo assim, nosso desafio é desenvolver um "controle remoto" que permita enviar solicitações para o robô executar algumas funções sem que a pessoa que está utilizando o controle remoto precise conhecer como o robô as executa. Vamos lá!

Codificando a solução com Command

Primeiro, vamos criar uma simples interface que terá apenas um método chamado `Executar`.

```
public interface ICommand
    {
        void Executar();
    }
```

Agora, vamos utilizar essa interface na criação de uma classe chamada `SimplesComando`. Veja o código:

```
public class SimplesComando : ICommand
    {
        private string _solicitacao = string.Empty;

        public SimplesComando(string solicitacao)
        {
            this._solicitacao = solicitacao;
        }

        public void Executar()
        {
            Console.WriteLine("Estou executando um Simples Comand
o de " + this._solicitacao);

        }
    }
```

Note que criamos uma variável privada chamada `_solicitacao`, que se inicia vazia, mas que, receberá seu valor quando essa classe for instanciada através do seu método construtor. E o método `Executar` escreve na tela uma mensagem concatenando com o conteúdo da variável `._solicitacao`.

E para ficar mais interessante, vamos criar uma classe chamada `ComplexoComando`, em que também vamos implementar a interface `ICommand`, que receberá mais de uma solicitação. Veja o código a seguir:

```
public class ComplexoComando : ICommand
    {
        private Receiver _receiver;
        private string _a;
        private string _b;

        public ComplexoComando(Receiver receiver, string a, strin
 b)
        {
            this._receiver = receiver;
            this._a = a;
            this._b = b;
        }

        public void Executar()
        {
            this._receiver.PrimeiroPedido(this._a);
            this._receiver.SegundoPedido(this._b);
        }
    }
```

Na classe `ComplexoComando`, criamos um método chamado `ComplexoComando`, que recebe duas variáveis (`_a` e `_b`) e um objeto do tipo `_receiver`, que vamos criar na sequência. A classe também conta com a implementação do método `Executar`, que executa o método `PrimeiroPedido` e `SegundoPedido` do objeto `_receiver`. Sendo assim, vamos criar a classe `Receiver`.

```
public class Receiver
    {
        public void PrimeiroPedido(string a)
        {
            Console.WriteLine("Comando Recebido: " + a);
        }

        public void SegundoPedido(string b)
        {
            Console.WriteLine("Outro Comando Recebido: " + b);
        }
    }
```

Na classe `Receiver` , criamos dois métodos que simplesmente retornam uma mensagem concatenando com o conteúdo da variável recebida do respectivo método. Sendo assim, podemos partir para a criação da classe que representará o nosso controle.

```
public class Controle
    {
        private ICommand comandoSimples;
        private ICommand comandoComplexo;

        public void EnviarComandoSimples(ICommand command)
        {
            this.comandoSimples = command;
        }

        public void EnviarComandoComplexo(ICommand command)
        {
            this.comandoComplexo = command;
        }

        public void Fazer()
        {
            Console.WriteLine("Ok, vou executar para você!");
            if (this.comandoSimples is ICommand)
                this.comandoSimples.Executar();

            if (this.comandoComplexo is ICommand)
                this.comandoComplexo.Executar();
        }
    }
```

Na classe `Controle`, criamos dois métodos (`EnviarComandoSimples` e `EnviarComandoComplexo`), ambos recebem como parâmetro uma interface `Command`. Já no método `Fazer` temos uma verificação para garantir que sejam executados os comandos equivalentes à classe solicitada, que pode ser a solicitação de um simples ou complexo comando.

Por fim, podemos finalizar nossa solução na classe `Program`, que ficará da seguinte forma:

```
class Program
    {
        static void Main(string[] args)
        {
            Controle controle = new Controle();
            controle.EnviarComandoSimples(new SimplesComando("Diz
er Oi!"));

            Receiver receiver = new Receiver();
            controle.EnviarComandoComplexo(new ComplexoComando(re
ceiver, "Abastecer o carro", "Calibrar os pneus do carro"));

            controle.Fazer();
            Console.ReadKey();
        }
    }
```

No método `Main` começamos instanciando `controle` e, em seguida, executamos o método `EnviarComandoSimples` passando como parâmetro uma nova instancia da classe `SimplesComando`, que terá como função executar o comando e emitir uma mensagem `Dizer Oi`. Na sequência, instanciamos `receiver`, que contém mais opções de execução de comandos (`PrimeiroPedido` e `SegundoPedido`) que são passados como parâmetro juntos com o `receiver` para a classe `ComplexoComando`. E, por fim, pedimos para que o controle execute tudo, por meio do método `Fazer`. Ao executar nossa

solução, veja o resultado a seguir.

```
Ok, vou executar para você!
Estou executando Comando de Dizer Oi!
Comando Recebido: Abastecer o carro
Outro Comando Recebido: Calibrar os pneus do carro
```

Figura 7.6: Resultado do Pattern Command

Note que foram exibidas as ações de acordo com as nossas solicitações. O mais legal é que as ações foram solicitadas ao controle e ele encarregou-se de executá-las. Veja uma ilustração das interações entre as classes que criamos na utilização do Design Pattern Command.

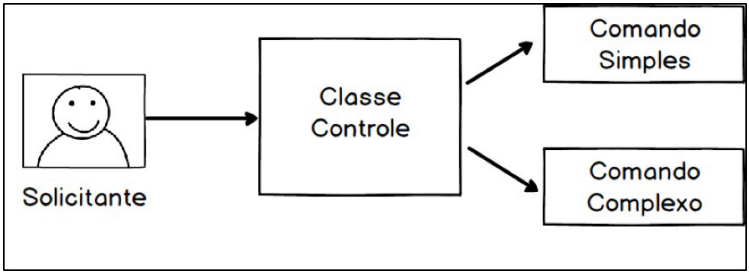


Figura 7.7: Ilustração do Command

Nosso solicitante é a nossa classe `Program`, pois é no método `Main` dela que utilizamos a classe `Controle`. Ela, por sua vez, é quem é capaz de interagir e executar os comandos das classes de comando simples e complexo. O solicitante nem precisar saber como as funções são implementadas, apenas solicita ao controle, que, de acordo com a função solicitada pelo cliente, encaminha para a respectiva classe. Sendo assim, mais um desafio foi concluído e você pode conhecer mais um Pattern. Parabéns!

7.6 STRATEGY

O Design Pattern Strategy é um padrão comportamental que permite configurar uma classe que seja capaz de ter muitos comportamentos. E para lhe mostrar e, claro, praticar esse padrão, vamos solucionar mais um desafio.

Hora do desafio

Entre os meus jogos preferidos do Super Nintendo, sem dúvidas está o Donkey Kong Country. Lembro-me de que os gráficos desse jogo eram incríveis e as fases do jogo eram muito envolventes e desafiadoras. O que também era legal neste jogo é que você poderia contar com a ajuda de outros animais encontrados dentro de caixas, de acordo com a fase do jogo, bastando libertá-los, como um avestruz, o peixe-espada, sapo e um papagaio.

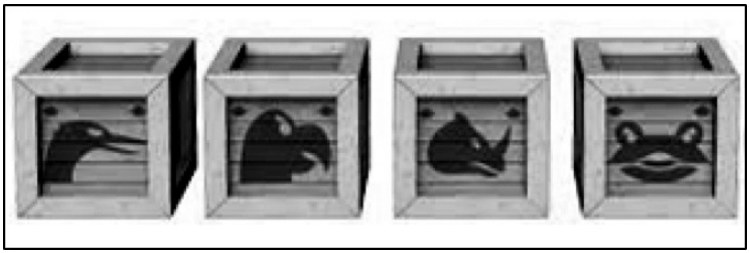


Figura 7.8: Ilustração das caixas de animais encontradas nas fases do jogo

Sendo assim, tomaremos esse cenário como exemplo, em que temos o desafio de criar uma classe flexível o suficiente que seja capaz de ter vários comportamentos. Ora seja de um sapo, ora de um avestruz etc. Então vamos lá!

Codificando a solução com Strategy

Começaremos a codificar a solução para o nosso desafio com o Design Pattern Strategy criando uma interface chamada `IAjuda`, que contará com um método chamado `Ajudar` que recebe um `pedido` como parâmetro com um retorno do tipo `string`.

```
namespace Strategy
{
    public interface IAjuda
    {
        string Ajudar(Ajuda pedido);
    }
}
```

Agora, vamos aplicar essa interface em uma nova classe chamada `Ajuda`.

```
namespace Strategy
{
    public class Ajuda
    {
        private IAjuda _IAjuda;

        public Ajuda(IAjuda ajuda)
        {
            _IAjuda = ajuda;
        }

        public string Ajudar()
        {
            return _IAjuda.Ajudar(this);
        }
    }
}
```

Na classe `Ajuda`, instanciamos a interface `IAjuda` que armazena o tipo de ajuda solicitado pelo método construtor dessa classe. Ainda na classe `Ajuda`, criamos o método chamado `Ajudar`, que retorna à execução do método do objeto

armazenado na interface `IAjuda`. Agora, vamos criar duas classes, uma chamada `Sapo` e outra `Papagaio`.

```
public class Sapo : IAjuda
    {
        public string Ajudar(Ajuda pedido)
        {
            return "Sou um sapo e posso ajudar você a pular bem a
lto!";
        }
    }

public class Papagaio : IAjuda
    {
        public string Ajudar(Ajuda pedido)
        {
            return "Sou um Papagaio e posso ajudar você a voar!";
        }
    }
```

Ambas as classes implementam a interface `IAjuda` e contam com um método chamado `Ajudar`, que recebe como parâmetro um `pedido`, e então retornam uma mensagem dizendo como podem ajudar de acordo com o seu comportamento. Por fim, podemos finalizar nossa codificação na classe `Program`.

```
class Program
    {
        static void Main(string[] args)
        {
            var papagaio = new Ajuda(new Papagaio());
            Console.WriteLine(papagaio.Ajudar());

            var sapo = new Ajuda(new Sapo());
            Console.WriteLine(sapo.Ajudar());

            Console.ReadKey();
        }
     }
```

Como você pode ver, criamos dois objetos, `papagaio` e

`sapo` , que instanciam a mesma classe chamada `Ajuda` . Na sequência, solicitamos a execução do método `Ajudar()` em ambos os objetos. E veja o resultado:

```
Sou um Papagaio e posso ajudar você a voar!
Sou um sapo e posso ajudar você a pular bem alto!
```

Figura 7.9: Resultado do Pattern Strategy

Sendo assim, note que a classe chamada `Ajuda` tem agora dois tipos com comportamentos distintos de ajuda no jogo (o sapo ou papagaio). Sendo assim, conseguimos implementar a proposta do Design ao desenvolver uma classe capaz de ter muitos comportamentos.

7.7 CHAIN OF RESPONSABILITY

A ideia do Design Pattern Chain of Responsability, ou, em português, Cadeia de Responsabilidade, é possibilitar a transferência de solicitações em uma cadeia de manipuladores. Esses manipuladores nada mais são do que objetos independentes, que, quando recebem uma solicitação, podem decidir se processarão a solicitação ou passarão para o próximo manipulador da cadeia. Vamos colocar esse padrão em prática em um novo desafio.

Hora do desafio

Um épico jogo de luta é o Marvel vs. Capcom disponível para diversas plataformas e versões que reúnem os heróis dos videogames e dos quadrinhos dos dois universos. Um recurso muito legal é a possibilidade de escolher mais de um personagem e,

durante a luta, trocar de personagem pressionado uma determinada sequência de botões, com a qual o lutador substituto já entra no ringue atacando.

Sendo assim, vamos imaginar que teremos a possibilidade de escolher três personagens e durante a luta queremos que cada personagem seja convocado de acordo com a quantidade de força e/ou poder que queremos naquele momento da batalha. Portanto esse será o nosso desafio!

Codificando a solução com Chain of Responsability

Começaremos a codificar nossa solução criando uma classe abstrata para definir um `Manipulador`.

```
public abstract class Manipulador
    {
        protected Manipulador sucessor;

        public void defineSucessor(Manipulador sucessor)
        {
            this.sucessor = sucessor;
        }

        public abstract void Convoca(int quantidadePoder);
     }
```

Assim, criamos uma classe abstrata com o nome `Manipulador` que conta com um método chamado `defineSucessor`. Em seguida, criamos outro método chamado `Convoca`, que tem como parâmetro apenas um valor do tipo inteiro que se refere à quantidade de poder.

Logo, podemos criar três personagens de exemplo que implementam a classe abstrata `Manipulador` que acabamos de criar.

```
public class PersonagemA : Manipulador
    {
        public override void Convoca(int quantidadePoder)
        {
            if (quantidadePoder >= 0 && quantidadePoder < 10)
                Console.WriteLine("{0} convocado para uma força d
e poder de {1}", this.GetType().Name, quantidadePoder);
            else if (sucessor != null)
                sucessor.Convoca(quantidadePoder);
        }
    }


public class PersonagemB : Manipulador
    {
        public override void Convoca(int quantidadePoder)
        {
            if (quantidadePoder >= 10 && quantidadePoder < 20)
                Console.WriteLine("{0} convocado para uma força d
e poder de {1}", this.GetType().Name, quantidadePoder);
            else if (sucessor != null)
                sucessor.Convoca(quantidadePoder);
        }
     }


public class PersonagemC : Manipulador
    {
        public override void Convoca(int quantidadePoder)
        {
            if (quantidadePoder >= 20 && quantidadePoder < 30)
                Console.WriteLine("{0} convocado para uma força d
e poder de {1}", this.GetType().Name, quantidadePoder);
            else if (sucessor != null)
                sucessor.Convoca(quantidadePoder);
        }
     }
```

Note que a estrutura das três classes é a mesma, contudo, elas diferem-se apenas no valor verificado pela estrutura de condição `if` . Caso a quantidade de poder solicitado enquadra-se no valor verificado, será exibida uma mensagem de qual personagem foi

convocado e a quantidade do poder solicitado. Porém, se a quantidade de poder solicitado não estiver dentro do que esse personagem é capaz, então, é convocado outro personagem passando a quantidade do poder necessário como parâmetro. Percebeu a relação entre os três personagens? Veja na ilustração a seguir.

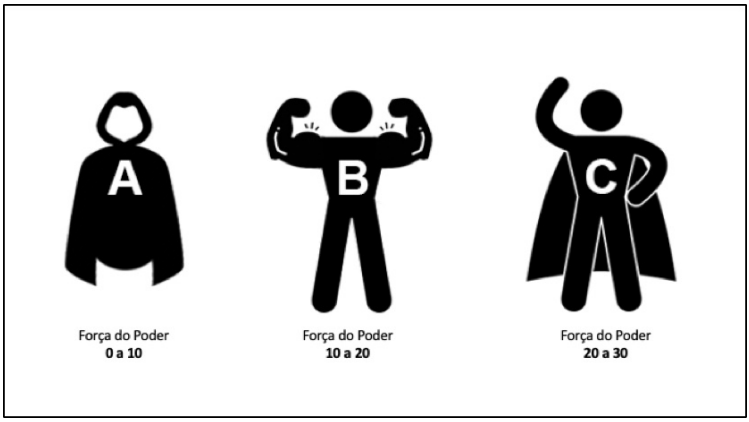


Figura 7.10: Ilustração do Pattern Chain of Responsability

Vamos à classe `Program` colocar em ação a aplicação do Design Pattern Chain of Responsability.

```
class Program
    {
        static void Main(string[] args)
        {
            Manipulador pA = new PersonagemA();
            Manipulador pB = new PersonagemB();
            Manipulador pC = new PersonagemC();

            pA.defineSucessor(pB);
            pB.defineSucessor(pC);

            int[] requisicoes = { 5, 8, 15, 20, 18, 3, 27, 20 };
```

```
            foreach (int req in requisicoes)
            {
                pA.Convoca(req);
            }

            Console.ReadKey();
        }
    }
```

Note que começamos criando um `Manipulador` para cada personagem (`pA` , `pB` e `pC`). Em seguida, definimos quem serão os sucessores, ou seja, caso o poder solicitado seja maior que a capacidade do personagem, ele passará a responsabilidade para outro personagem. Logo, criamos um `array` com alguns números que representam uma lista de solicitações de quantidade de poderes ao longo do jogo. Em seguida, utilizando um laço de repetição *foreach*, percorremos a lista que contém alguns números (que representam uma quantidade de poder solicitado ao personagem). Repare que é o personagem chamado `pA` (Personagem A) quem dispara o método `Convoca` , que tem como parâmetro `req` referente à quantidade de poder solicitado. Caso consiga, vai executar; caso o poder solicitado seja maior que sua capacidade, ele mesmo vai passar a responsabilidade para outro personagem (no caso, o `pB`). Logo, ao executar, temos o resultado a seguir.

```
PersonagemA convocado para uma força de poder de 5
PersonagemA convocado para uma força de poder de 8
PersonagemB convocado para uma força de poder de 15
PersonagemC convocado para uma força de poder de 20
PersonagemB convocado para uma força de poder de 18
PersonagemA convocado para uma força de poder de 3
PersonagemC convocado para uma força de poder de 27
PersonagemC convocado para uma força de poder de 20
```

Figura 7.11: Resultado do Pattern Chain of Responsability

Desde modo, conseguimos criar uma cadeia de personagens que transferem em tempo de execução do programa a responsabilidade para o próximo personagem caso a quantidade de poder solicitada seja maior que a sua capacidade. Legal, não é mesmo? Sendo assim, mais um desafio concluído. Parabéns!

7.8 ITERATOR

A ideia deste padrão de projetos, o **Design Pattern Iterator**, é simplesmente organizar e/ou agregar objetos em uma coleção, também permitindo percorrer essa coleção. Vamos ao desafio!

Hora do desafio

Tomaremos como exemplo um jogo de fases tal como o Super Mario World. O que é interessante nesse estilo de jogo é que há um mapa de fases que nos dá opções de escolher algumas fases (seguir fases para direita ou para esquerda, por exemplo). Sendo assim, precisamos criar uma coleção que armazenará as fases do nosso jogo, onde cada fase será um objeto independente (visto que cada fase do jogo pode ter suas particularidades) e, quando necessário, percorrer a coleção de fases. Vamos ao código!

Codificando a solução com Iterator

Vamos iniciar a codificação do nosso projeto com a criação de uma interface que vai representar o nosso Iterator definindo os métodos para acessar e percorrer nossa coleção de fases.

```
public interface IIterator
    {
        string primeiroItem { get; }
        string proximoItem { get; }
```

```
        string atualItem { get; }
        bool estaPronto { get; }
    }
```

Agora, vamos criar uma outra interface chamada `IAggregate`, que terá o método para criar um objeto iterador quando necessário.

```
public interface IAggregate
    {
        IIterator GetIterator();
        string this[int indexItem] { set; get; }
        int Contador { get; }
    }
```

Neste momento, já temos as interfaces necessárias. Agora, podemos criar a classe responsável pela coleção de fases do nosso jogo, chamada `Aggregate`, que vai implementar a interface `IAggregate`.

```
public class Aggregate : IAggregate
    {
        List<string> colecao = null;

        public Aggregate()
        {
            colecao = new List<string>();
        }

        public IIterator GetIterator()
        {
            return new Iterator(this);
        }

        public string this[int indexItem]
        {
            get
            {
                if (indexItem < colecao.Count)
                {
                    return colecao[indexItem];
                }
```

```
            else
            {
                return string.Empty;
            }
        }
        set
        {
            colecao.Add(value);
        }
    }

    public int Contador
    {
        get
        {
            return colecao.Count;
        }
    }
}
```

Note que começamos com a criação de uma lista com o nome `colecao` que se inicia nula, e o método construtor da classe `Aggregate` instancia esta lista. Em seguida, temos o método `GetIterator` que instancia um objeto `Iterator` e passa como parâmetro a própria classe `Aggregate` por meio da palavra reservada *this*.

Na sequência, criamos um atributo do tipo `string` que faz referência à própria classe com o uso da palavra *this*. Esse atributo tem um método `get` que verifica se o `indexItem` solicitado existe dentro da coleção para retornar a fase do jogo armazenada na posição correspondente em nossa coleção. Caso maior, retorna uma `string` vazia, ou seja, a fase não existe na coleção de fases do jogo.

Ainda, temos logo abaixo o método `set` , que adiciona uma nova fase na coleção, e por fim criamos o método `Contador` , que

apenas retorna a quantidade de fases que a coleção possui.

Desta forma, temos a estrutura do Design Pattern Iterator pronta. Vamos então implementá-la na classe `Program` no método principal de nossa aplicação.

```
class Program
    {
        static void Main(string[] args)
        {
            Aggregate colecaoDeFases = new Aggregate();

            colecaoDeFases[0] = "Fase 1";
            colecaoDeFases[1] = "Fase 2";
            colecaoDeFases[2] = "Fase 3";
            colecaoDeFases[3] = "Fase 4";
            colecaoDeFases[4] = "Fase 5";

            IIterator itterator = colecaoDeFases.GetIterator();

            for (string s = itterator.primeiroItem; itterator.est
aPronto == false; s = itterator.proximoItem)
            {
                Console.WriteLine(s);
            }

            Console.ReadKey();
        }
```

Começamos pela criação da nossa coleção, que é do tipo `Aggregate`, e atribuímos cinco fases em nosso jogo (fique à vontade para adicionar quantas fases você achar interessante). Na sequência, criamos o nosso objeto `iterator` e utilizamos seus métodos para percorrer a nossa coleção em um laço de repetição. Ao executar, veja o resultado.

```
Fase 1
Fase 2
Fase 3
Fase 4
Fase 5
```

Figura 7.12: Resultado do Pattern Iterator

Sendo assim, conseguimos adicionar novas fases e percorrer a coleção exibindo todas as fases do jogo utilizando o padrão de projetos Iterator. Parabéns!

7.9 MEDIATOR

Este Design Pattern, o **Mediator**, é um padrão de projeto comportamental bem interessante. Basicamente, seu objetivo é impedir a comunicação direta entre objetos, sendo que o Mediator, ou mediador em português, é quem fica responsável por receber e distribuir as comunicações entre os objetos.

Para exemplificar, pense no Mediator como um carteiro, pois as cartas (mensagens) estão nas mãos dele, e é ele quem fica responsável em entregar cada carta para a casa correta.

Figura 7.13: Ilustração do Pattern Mediator

Hora do desafio

Uma das funções que os jogos modernos disponibilizam é a possibilidade de jogar online e a troca de mensagens entre os jogadores durante o jogo, algo que não era possível um tempo atrás nos primeiros consoles.

Sendo assim, o nosso desafio será desenvolver uma solução que permita a troca de mensagens entre jogadores. Para tanto, utilizaremos o Design Pattern Mediator para receber e enviar as mensagens aos jogadores corretamente, tal como um carteiro que recebe a carta de um remetente e entrega ao respectivo destinatário. Vamos lá!

Codificando a solução com Mediator

Começaremos pelo desenvolvimento uma classe abstrata para representar o jogador.

```
public abstract class Jogador
    {
        protected Mediador mediador;

        public Jogador(Mediador mediador)
        {
            this.mediador = mediador;
        }
    }
```

Em seguida, vamos criar dois jogadores, que serão o suficiente para estabelecer uma troca de mensagens.

```
public class Jogador1 : Jogador
    {
        public Jogador1(Mediador mediador) : base(mediador) {}

        public void Enviar(string mensagem)
```

```
        {
            mediador.Enviar(mensagem, this);
        }

        public void Notificar(string mensagem)
        {
            Console.WriteLine("Mensagem do Jogador 1: " + mensage
m);
        }
    }


public class Jogador2 : Jogador
    {
        public Jogador2(Mediador mediador) : base(mediador) {}

        public void Enviar(string mensagem)
        {
            mediador.Enviar(mensagem, this);
        }

        public void Notificar(string mensagem)
        {
            Console.WriteLine("Mensagem do Jogador 2: " + mensage
m);
        }
    }
```

Note que ambos os jogadores herdam a classe abstrata `Jogador`. No método construtor, tanto `Jogador1` quanto `Jogador2` esperam receber um `Mediador` para que fique definido quem media as mensagens desse jogador. Além do mais, ambos contam com um método `Enviar` e `Notificar`.

O método `Enviar` é fundamental aqui, pois é ele que envia a mensagem para o mediador que definirá seu destino. Sendo assim, vamos criar a classe abstrata `Mediador`.

```
public abstract class Mediador
    {
        public abstract void Enviar(string mensagem, Jogador joga
```

```
dor);
    }
```

E agora, podemos implementar o `Mediador` de uma forma concreta, que será o cerne da aplicação.

```
public class MediadorConcreto : Mediador
    {
        private Jogador1 j1;
        private Jogador2 j2;

        public Jogador1 Jogador1
        {
            set { j1 = value; }
        }

        public Jogador2 Jogador2
        {
            set { j2 = value; }
        }

        public override void Enviar(string mensagem, Jogador joga
dor)
        {
            if (jogador == j2)
                j1.Notificar(mensagem);
            else
                j2.Notificar(mensagem);
        }
    }
```

Na codificação dessa classe iniciamos com a definição de dois jogadores. Em seguida, definimos que seus respectivos métodos `set` vão retornar apenas o seu próprio valor. Por fim, temos o método `Enviar`, que espera como parâmetro a mensagem e o jogador que deverá recebê-la, e valida se o jogador destinatário é jogador um (`j1`) ou jogador dois (`j2`).

O padrão Mediator está pronto. Agora, vamos utilizá-lo em nossa classe `Program`.

```
class Program
    {
        static void Main(string[] args)
        {
            MediadorConcreto mediador = new MediadorConcreto();

            Jogador1 j1 = new Jogador1(mediador);
            Jogador2 j2 = new Jogador2(mediador);

            mediador.Jogador1 = j1;
            mediador.Jogador2 = j2;

            j1.Enviar("Essa partida foi muito boa!");
            j2.Enviar("Foi sensacional. Conseguimos uma ótima pon
tuação!");

            Console.ReadKey();
        }
    }
```

Na classe `Program`, começamos com a definição de quem será nosso `mediador` e, na sequência, definimos os dois jogadores passando para ambos o mesmo `mediador`. Por fim, por meio do jogador um (`j1`) enviamos uma mensagem e fazemos o mesmo com o jogador dois (`j2`). Ao executar o programa podemos visualizar o seguinte resultado.

```
Mensagem do Jogador 2: Essa partida foi muito boa!
Mensagem do Jogador 1: Foi sensacional. Conseguimos uma ótima pontuação!
```

Figura 7.14: Resultado do Pattern Mediator

Você conseguiu implementar e utilizar o padrão Mediator. Pois todas as mensagens são enviadas a um mediador que fica responsável por entregá-las aos jogadores.

7.10 MEMENTO

O **Design Pattern Memento** é um padrão comportamental que simplesmente nos dá a possibilidade de salvar o estado de um objeto e restaurá-lo quando necessário.

Hora do desafio

Um recurso que eu acredito que você já tenha utilizado no videogame é o de pausar o jogo, para atender o telefone, por exemplo; e em seguida bastou pressionar o botão play do seu controle para retomar o jogo do exato momento em que você parou. Logo, este será o nosso desafio. Criar uma aplicação que permita um objeto guardar o ponto em que parou e retomar quando solicitado, e claro, para isso, utilizaremos o padrão de projetos Memento.

Codificando a solução com Memento

Iniciaremos a codificação pela classe que levará o nome do Design Pattern que estamos explorando aqui, o Memento.

```
public class Memento
    {
        private string _estado;

        public Memento(string estado)
        {
            this._estado = estado;
        }

        public string Estado
        {
            get { return _estado; }
        }
    }
```

Sempre que instanciada a classe `Memento`, o método construtor vai se encarregar de atribuir um estado, que, em nosso exemplo, será o de `pause` - que representa o jogo quando estiver parado - e o de `play` - para quando o jogo estiver em ação. Agora, criaremos uma outra classe de nome bastante pertinente ao seu propósito, que é `Armazena`, para nos ajudar a armazenar e a recuperar os estados de um objeto.

```
public class Armazena
    {
        public Memento Memento { get; set; }

    }
```

Agora, a classe que vamos criar é a `Acao`, que conterá a lógica de controle dos estados do jogo. Veja como fica o código a seguir.

```
public class Acao
    {
        private string _estado;

        public string Estado
        {
            get { return _estado; }
            set
            {
                _estado = value;
                Console.WriteLine("Estado do Jogo = " + _estado);
            }
        }

        public Memento CriarMemento()
        {
            return (new Memento(_estado));
        }

        public void RestaurarEstado(Memento memento)
        {
            Console.WriteLine("Restaurando o estado...");
            Estado = memento.Estado;
        }
```

```
    }
```

Na classe `Acao` temos um atributo `_estado`, um método `CriarMemento` que armazena o estado do objeto e outro método chamado `RestaurarEstado`, que resgata o estado anterior do objeto que foi armazenado por meio do `memento`. Sendo assim, vamos à classe `Program` para fazer uso desse Design Pattern.

```
class Program
    {
        static void Main(string[] args)
        {
            Acao acao = new Acao();
            acao.Estado = "play";

            Armazena armazena = new Armazena();
            armazena.Memento = acao.CriarMemento();

            acao.Estado = "pause";
            acao.RestaurarEstado(armazena.Memento);

            Console.ReadKey();
        }
    }
```

Começamos com a instanciação de `acao` e atribuímos inicialmente o estado de `play`, que significa que o jogo está em ação. Guardamos o seu estado utilizando a classe `armazena`, que por meio do `Memento` armazena o estado atual do objeto. Em seguida, atribuímos um novo estado ao jogo, agora o de `pause`, que significa que o jogo está parado neste momento. E, por fim, executamos o método `RestaurarEstado` que retoma o estado anterior do `pause`, que é o de `play` e, assim, o jogo volta à ação. Ao executar a aplicação, você pode visualizar o seguinte resultado.

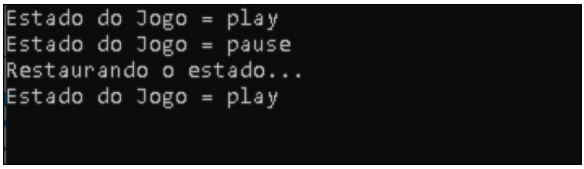


Figura 7.15: Resultado do Pattern Memento

Legal, não é mesmo? Espero você no próximo e último Design Pattern da nossa saga de padrões de projetos comportamentais.

7.11 STATE

O State é o último Design Pattern da nossa saga dos Design Patterns de Comportamento. O objetivo desse padrão de projeto é permitir que um objeto mude o seu comportamento quando o seu estado interno é alterado.

Talvez, você tenha notado uma semelhança com o Pattern Strategy, mas há uma diferença fundamental. No padrão State, os estados particulares podem estar cientes um do outro e iniciar transições de um estado para outro, enquanto no padrão Strategy há muitos comportamentos e que quase nunca se conhecem.

Logo, convido você a um desafio, onde teremos como cenário um jogo clássico para aplicarmos e compreendermos o padrão State. Vamos lá!

Hora do desafio

No clássico jogo Donkey Kong Country do Super Nintendo, era possível jogar as fases do jogo com dois personagens, contudo, apenas um deles era controlado enquanto o outro apenas seguia o

definido como personagem principal. Mas o legal é que, a qualquer momento do jogo, era possível mudar o comportamento do personagem principal invertendo a situação. E é nesse contexto que vamos aplicar o padrão State para alterar o comportamento de um objeto. Vamos lá!

Codificando a solução com State

Primeiro, vamos criar a classe `State`, que possuirá o método `Acao` que espera como parâmetro `Contexto`.

```
public abstract class State
    {
        public abstract void Acao(Contexto contexto);
    }
```

Na classe `Contexto`, o método construtor registra o `estado` que, quando alterado pelo modificador `set`, retorna qual é o comportamento que o objeto assumiu. Por fim, o método `Trocar` envia a própria classe por meio do uso da palavra reservada `this` como nova ação.

```
public class Contexto
    {
        private State _estado;

        public Contexto(State estado)
        {
            this._estado = estado;
        }

        public State State
        {
            get { return _estado; }
            set
            {
                _estado = value;
                Console.WriteLine("Agora o comportamento é de: "
+ _estado.GetType().Name);
```

```
            }
        }

        public void Troca()
        {
            _estado.Acao(this);
        }
    }
```

Deste modo, podemos criar os dois personagens que implementam a classe `State`.

```
public class PersonagemA : State
    {
        public override void Acao(Contexto contexto)
        {
            contexto.State = new PersonagemB();
        }
    }

public class PersonagemB : State
    {
        public override void Acao(Contexto contexto)
        {
            contexto.State = new PersonagemA();
        }
    }
```

Note que ambas as classes de personagens (`PersonagemA` e `PersonagemB`) possuem o método `Acao` , que espera como parâmetro `Contexto` e define um novo personagem. Quando solicitada a troca, `PersonagemA` recebe `PersoangemB` e vice-versa.

Agora, vamos à classe `Program` e instanciar `Contexto` passando `PersonagemA` como parâmetro. Em seguida, vamos solicitar quatro trocas.

```
class Program
    {
        static void Main(string[] args)
```

```
        {
            Contexto contexto = new Contexto(new PersonagemA());

            contexto.Troca();
            contexto.Troca();
            contexto.Troca();
            contexto.Troca();

            Console.ReadKey();
        }
    }
```

Ao executar o programa, você notará no resultado que o comportamento de `PersoangemA` mudou todas as vezes em que solicitamos a troca.

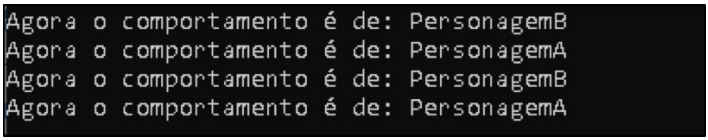

Agora o comportamento é de: PersonagemB
Agora o comportamento é de: PersonagemA
Agora o comportamento é de: PersonagemB
Agora o comportamento é de: PersonagemA

Figura 7.16: Resultado do Pattern State

Sendo assim, encerramos a saga dos Design Pattern Comportamentais. Parabéns por chegar até aqui!

Conclusão

Neste capitulo, você trabalhou com os **Design Patterns com o propósito comportamental**. Iniciamos com o padrão **Template Method**, que define um modelo de passos abstratos que serão fornecidos às suas subclasses.

Em seguida, você desenvolveu o padrão **Interpreter**, assim como o seu próprio nome diz, torna um objeto capaz de interpretar uma representação (conjunto de regras).

Na sequência, o padrão da vez foi o **Observer**, que permite que objetos sejam informados sobre mudanças ocorridas em outros objetos.

Em seguida, estudamos o Pattern **Visitor**, que permite adicionar novos comportamentos a um objeto existente, sem precisar alterar sua estrutura.

Também trabalhamos com o padrão **Command**, que permite transformar uma solicitação em um objeto que conterá todas as informações sobre ela.

Partimos para o **Strategy**, que permite configurar uma classe que seja capaz de ter muitos comportamentos.

Depois, o foco foi o padrão **Chain of Responsability**, que possibilita a transferência de solicitações em uma cadeia de manipuladores. Já o padrão **Iterator** tem como objetivo simplesmente organizar objetos em uma coleção.

Logo em seguida, o foco foi o Pattern **Mediator**, que impede a comunicação direta entre objetos.

O padrão **Memento**, como vimos, é um padrão comportamental que simplesmente nos dá a possibilidade de salvar o estado de um objeto e restaurá-lo quando necessário.

E, por fim, para encerrar essa saga intensa, trabalhamos com o padrão **State**, que permite alterar o comportamento de um objeto quando solicitado.

Desta forma, encerramos mais um capítulo de muita prática e você tem uma ótima bagagem sobre os **Design Patterns Comportamentais**. Parabéns!

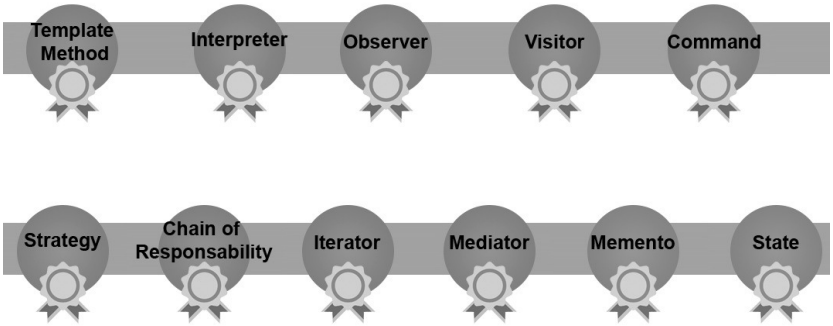


Figura 7.17: A saga dos Design Patterns Comportamentais

Capítulo 8

CONCLUSÃO

Primeiramente, parabéns por chegar até aqui! Foi um percurso e tanto, em que você explorou muitos Design Patterns e superou desafio a desafio proposto neste livro, além dos que naturalmente tenham surgido pelo caminho. Portanto, parabéns!

Agora, você tem em sua bagagem de desenvolvimento um amplo conhecimento sobre padrões de projeto de software, sobretudo, soluções reutilizáveis de software orientado a objetos aos quais você poderá recorrer para aplicar em seus projetos.

Contudo, eu recomendo que você não pare por aqui e continue estudando. A todo o momento novos padrões de projetos e/ou tendência surgem e, com isso, todos nós somos eternos aprendizes. Mas, fato é que grande parte dos padrões recentes tem como referência os padrões que você aprendeu neste livro, o que vai facilitar e ajudar no aprendizado de novos padrões.

Por isso, reitero, continue estudando e experimentando novos padrões e expandindo os seus horizontes, tal como você já fez aqui. E bons estudos!